Trace-Based Control-Flow Analysis

Benoît Montagu, Thomas Jensen

Trace-Based Control-Flow Analysis

Benoît Montagu
Inria
Rennes, France
benoit.montagu@inria.fr

Thomas Jensen
Inria
Rennes, France
thomas.jensen@inria.fr

Abstract

We define a small-step semantics for the untyped λ-calculus, that traces the β-reductions that occur during evaluation. By abstracting the computation traces, we reconstruct k-CFA using abstract interpretation, and justify constraint-based k-CFA in a semantic way. The abstract interpretation of the trace semantics also paves the way for introducing widening operators in CFA that go beyond existing analyses, that are all based on exploring a finite state space. We define ∇CFA, a widening-based analysis that limits the cycles in call stacks, and can achieve better precision than k-CFA at a similar cost.

CCS Concepts: **• Software and its engineering** → **Automated static analysis**; **• Theory of computation** → *Operational semantics*; **Program analysis**; *Abstraction*; Functional constructs.

Keywords: λ-calculus, control flow analysis, program traces, abstract interpretation, widening

ACM Reference Format:
Benoît Montagu and Thomas Jensen. 2021. Trace-Based Control-Flow Analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454057

1 Introduction

Control-flow analysis (CFA) [25, 42] is a family of static analyses whose goal is to determine which functions can be called at a given program point. For this purpose, CFAs generally compute two maps: a *cache* that determines the closures that each program point may evaluate to, and an *environment* that describes the values that could be assigned to each variables. With this information, it becomes possible to inline functions, and to remove safely some dynamic tag checks that must be performed in dynamically-typed languages. CFA of higher-order or object-oriented languages is difficult, since the control flow graph of a program is not static. Thus, CFAs try to compute over-approximations of the dynamic call graph. In the functional language community, two main research tracks on CFAs have emerged.

A first group of work has employed *constraints* to model the conditions under which a solution to a CFA problem is correct. Work in this vein [13, 16, 31, 33, 46] transforms a given program into a logical constraint—usually set-based constraints—that must be solved by a second program—the constraint solver. We refer to this research as "constraint-based analysis" (CBA).

A second group of work has focused on analysing *abstract machines*, and devised abstract interpretations of (small-step) abstract-machine-based semantics [15, 18, 28, 28]. In this "abstracting abstract machines" (AAM) research track, the low-level elements of abstract machines—environments, stores, continuation stacks—must be abstracted. A related approach has been applied to definitional (big-step) interpreters [4, 9].

The methodology of abstract interpretation [3] has been successfully applied to the AAM approach to justify the soundness of AAM-based analyses. The CBA approach, on the other hand, has focused on syntactic techniques, such as preservation properties, similar to the syntactic type soundness technique. More semantic justifications for the CBA approach, based on abstract interpretation, are still missing.

Although they differ in many ways, the AAM and CBA families have something in common: to ensure the termination of their analyses, they both perform approximations so that their search space becomes finite. Based on the finiteness of the search space, they are able to prove complexity results for CFAs [43]. This finiteness, however, impedes the use of abstract domains with unbounded heights. One such domain is the interval domain, that abstracts sets of integers. The restriction to a finite search space has also diminished the attention to *widening techniques* [3] for CFAs. While widening is often necessary to ensure the termination of analysers, it is also beneficial in finite domains, where it accelerates the convergence to a solution.

In this paper, we seek to reconcile CFA with abstract interpretation in two ways. First, we show that the control-flow analysis of a λ-term is an abstraction of its execution trace. This trace records two kinds of events: which function calls occurred, and which values were returned at a program point. This suffices to reconstruct the *caches* and *environments* found in CFAs. We designed the semantics so that the events track the contexts in which they are produced (at which program point, and after which function call). We

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454057

show how to recover the usual notion of *call strings* [40] from this contextual information. This provides a formalisation of the folklore knowledge that "context-sensitivity is trace partitioning". This process of building traces and abstracting them is general [2] and can be applied with benefits to other static analyses. Based on this abstraction, we give a *modular* soundness proof of the constraint-based approach.

In a second stage, we derive by abstract interpretation the ∇CFA analysis, an expressive control-flow analysis that is built on a *infinite* domain, and that employs widening to ensure convergence. The precision of this analysis indicates that further research on widening techniques for CFAs is worthwhile.

The contributions of the paper are organised as follows:

- We define a simple, substitution-based small-step semantics for the λ-calculus, that tracks the values produced in each evaluation contexts and the β-reductions that are performed during reduction (§2).
- We define a stack of abstractions (§3) from which we build the global environment and cache found in standard CFAs [33] out of the computation trace of a program. As a methodology, we derive for each abstraction layer the constraints that are satisfied by the abstracted semantics. At the end of our abstraction stack, we obtain a family of CFAs that generalise k-CFA, and that is parameterised by context sensitivity policies.
- Using the previous abstractions, we define a semantic criterion for the solutions of a constraint-based CFA problem. Using this criterion, we give a novel, *modular* proof of soundness for the constraint rules of k-CFA (§4), where each rule is verified separately. This gives *a semantic justification for constraint-based CFA*.
- Based on our semantic development, we present ∇CFA (§5), a novel control-flow analysis that goes beyond the finite state abstraction used in traditional CFAs. ∇CFA employs a widening operator, that is used at specific points, that are determined by the k^* heuristic by limiting call strings to "at most k cyclic call sites". It can replace the "last k call sites" heuristic from k-CFA.
- Based on the implementation of the analysis, we discuss experimental results (§6), by comparing 0-CFA and 1-CFA with our new analysis and heuristics on a set of example programs taken from the literature. The new analysis takes advantage of non-finite integer domains, such as intervals, to compute more precise approximations with a similar computational cost.

2 A Labelled λ-Calculus

2.1 Syntax

We assume the existence of three infinite sets $\mathbb{X}$, $\mathbb{L}$ and $\mathbb{P}$. The set of variables $\mathbb{X}$ contains *variables*, that are denoted by the metavariables x, y, z. The set $\mathbb{L}$ contains *labels*, that are denoted by the metavariable ℓ. Labels are put on lambda abstractions and serve to identify *binding sites* of variables. The set $\mathbb{P}$ contains *program points* (also called *locations*), that are denoted by the metavariable p.

Definition 2.1 (Syntax)**.** The set of terms $\mathcal{T}$ is inductively defined as follows:

$$\begin{array}{rcll} t \in \mathcal{T} & ::= & \mathsf{x} & \text{(Variables)} \\ & | & \lambda^{\ell}\mathsf{x}.\, t & \text{(Abstraction)} \\ & | & t\, t & \text{(Application)} \\ & | & [t]^{a} & \text{(Annotation)} \end{array}$$

The terms of the language are the terms of the untyped λ-calculus extended with the annotation construct $[t]^a$, that has no computational meaning, but is used to collect information about the evaluation of t using the annotation a.

We introduce two kinds of annotations: *program points* $p \in \mathbb{P}$ and *function calls* $\mathsf{Call}(\lambda^{\ell}\mathsf{x}.\, t, v)$. Program point annotations will serve to record (syntactic) call sites, and function call annotations to keep track of the terms actually applied in the call. Annotation paths π are sequences of annotations, and will serve as *contexts*. For example, a term $[(\lambda^{\ell}\mathsf{x}.\, t)\, v]^p$ reduces to $[[t[\mathsf{x} \leftarrow v]]^{\mathsf{Call}(\lambda^{\ell}\mathsf{x}.\, t, v)}]^p$, and any subsequent reduction will happen in a context of the form $\pi = p \cdot \mathsf{Call}(\lambda^{\ell}\mathsf{x}.\, t, v) \cdot \pi'$. This context tells us that the call site of the actual call is p. We let δ range over *call site paths i.e.* those annotation paths that only consist of program points. We write $\pi_1 \cdot \pi_2$ for the concatenation of paths.

Definition 2.2 (Annotations and paths)**.**

$$\begin{array}{rclcll} a & \triangleq & p & | & \mathsf{Call}(\lambda^{\ell}\mathsf{x}.\, t, v) & \text{(Annotations)} \\ \pi & \triangleq & \varepsilon & | & \pi a & \text{(Annotation paths)} \\ \delta \in \mathbb{D} & \triangleq & \varepsilon & | & \delta p & \text{(Call site paths)} \end{array}$$

The set of free variables $\operatorname{fv} t$ of a term t follows the standard definition. We write $t[\mathsf{x} \leftarrow u]$ for the term t where the variable x is replaced by the term u (without capture of variables). We omit its definition. The set of values $\mathbb{V}$ is standard for a call by value λ-calculus.

Definition 2.3 (Values). $v \in \mathbb{V} \quad ::= \quad \lambda^{\ell}\mathsf{x}.\, t$

2.2 Small-Step Semantics

The small-step semantics is a mostly standard call by value reduction semantics, but extended with *events* that are emitted at certain reduction steps. A β_v-redex is reduced by substituting the actual argument for the formal argument, and by emitting an event $\beta(\pi, \lambda^{\ell}\mathsf{x}.\, t, v)$ signalling that a β-reduction has been performed. Similarly, the reduction of a value under an annotation is done by discarding the annotation and returning the value. This reduction emits an event $\mathsf{Ret}(\pi, a, v)$ recording that the value was returned at that annotation (program point or function call). The path component π of an event will be used to keep track of the execution history that led to the emission of that event—we explain its use in more detail below. The set of *traces* $\mathbb{T}$ is the set of finite (possibly empty) sequences of events. We write $\mathrm{tr}_1 \mathbin{+\!\!+} \mathrm{tr}_2$ for the concatenation of traces.

Definition 2.4 (Events and traces)**.**

$$\begin{array}{rcll} \mathrm{e} & \triangleq & \mathsf{Ret}(\pi, a, v) \quad \mid \quad \beta(\pi, \lambda^{\ell}\mathrm{x}.\, t, v) & \text{(Events)} \\ \mathrm{tr} \in \mathbb{T} & \triangleq & \varepsilon \quad \mid \quad \mathrm{e}, \mathrm{tr} & \text{(Traces)} \end{array}$$

The operational semantics uses the following operator for prefixing all events in a trace with an annotation path π.

Definition 2.5 (Prefixing by a path)**.**

$$\begin{array}{rcl} \pi \cdot \mathsf{Ret}(\pi', a, v) & \triangleq & \mathsf{Ret}(\pi \cdot \pi', a, v) \\ \pi \cdot \beta(\pi', \lambda^{\ell}\mathrm{x}.\, t, v) & \triangleq & \beta(\pi \cdot \pi', \lambda^{\ell}\mathrm{x}.\, t, v) \\ \pi \cdot \varepsilon & \triangleq & \varepsilon \\ \pi \cdot (\mathrm{e}, \mathrm{tr}) & \triangleq & (\pi \cdot \mathrm{e}), (\pi \cdot \mathrm{tr}) \end{array}$$

The notation $t \xrightarrow{\mathrm{e}} u$ denotes that the term t reduces in one step to u, and during this reduction, produces the event e. The definition of the one-step reduction follows:

$$(\lambda^{\ell}\mathrm{x}.\, t)\, v \xrightarrow{\beta(\varepsilon,\lambda^{\ell}\mathrm{x}.\,t,v)} [t[\mathrm{x} \leftarrow v]]^{\mathsf{Call}(\lambda^{\ell}\mathrm{x}.\,t,v)}$$

$$[v]^a \xrightarrow{\mathsf{Ret}(\varepsilon,a,v)} v \qquad \frac{t \xrightarrow{\mathrm{e}} t'}{t\,u \xrightarrow{\mathrm{e}} t'\,u} \qquad \frac{t \xrightarrow{\mathrm{e}} t'}{v\,t \xrightarrow{\mathrm{e}} v\,t'} \qquad \frac{t \xrightarrow{\mathrm{e}} u}{[t]^a \xrightarrow{a\cdot\mathrm{e}} [u]^a}$$

As we mentioned earlier, the contraction of a β_v-redex performs a substitution and encloses the reduct in an annotation, so as to remember that the reductions that will follow happen in the context of a function call. This is indeed what is meant by the contextual rule for annotations: any event that is emitted under an annotation is reported by prefixing that event with the annotation. Thus, annotations record the *contexts* of reductions.

The reflexive transitive closure is written $t \overset{\mathrm{tr}}{\longrightarrow}{}^{*} u$, and defined as follows:

$$t \overset{\varepsilon}{\longrightarrow}{}^{*} t \qquad\qquad \frac{t_1 \xrightarrow{\mathrm{e}} t_2 \qquad t_2 \overset{\mathrm{tr}}{\longrightarrow}{}^{*} t_3}{t_1 \overset{\mathrm{e},\mathrm{tr}}{\longrightarrow}{}^{*} t_3}$$

The rule for reflexivity traces no event, and the rule for transitivity collects the events in the order they occur.

Lemma 2.6. *The following inference rules are valid:*

$$\frac{t_1 \overset{\mathrm{tr}_1}{\longrightarrow}{}^{*} t_2 \quad t_2 \overset{\mathrm{tr}_2}{\longrightarrow}{}^{*} t_3}{t_1 \overset{\mathrm{tr}_1 +\!\!+ \mathrm{tr}_2}{\longrightarrow}{}^{*} t_3} \qquad\qquad \frac{t \overset{\mathrm{tr}}{\longrightarrow}{}^{*} u}{[t]^a \overset{a\cdot\mathrm{tr}}{\longrightarrow}{}^{*} [u]^a}$$

We give an example of term reduction in Fig. 1. The first step in the evaluation reduces the term annotated with 1 which returns a function. Similarly, the second step evaluates the term annotated with 5 to produce the argument to the function. The third step is the β-reduction that applies the function to the argument, emitting a β-event which states that a β-reduction occurred in the context 0. The two following return steps illustrate how the same lambda term $\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6$ can be returned under different annotations (here, 3 and 4), thereby distinguishing when it is used as a function and as an argument. The context in those two return events is the path $0 \cdot C_1 \cdot 2$ which contains the information that the returned value was obtained by first evaluating the call site annotated with 0, thereby initiating the call C_1, which in turn led to the call site labelled 2.

$$\begin{array}{rl} & [[\lambda^{\ell_1}\mathrm{x}.\,[[\mathrm{x}]^3[\mathrm{x}]^4]^2]^1\ [\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^5]^0 \\ \xrightarrow{\mathsf{Ret}(0,1,\lambda^{\ell_1}\mathrm{x}.\,[[\mathrm{x}]^3[\mathrm{x}]^4]^2)} & [(\lambda^{\ell_1}\mathrm{x}.\,[[\mathrm{x}]^3[\mathrm{x}]^4]^2)\ [\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^5]^0 \\ \xrightarrow{\mathsf{Ret}(0,5,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [(\lambda^{\ell_1}\mathrm{x}.\,[[\mathrm{x}]^3[\mathrm{x}]^4]^2)\ (\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)]^0 \\ \xrightarrow{\beta(0,\lambda^{\ell_1}\mathrm{x}.\,[[\mathrm{x}]^3[\mathrm{x}]^4]^2,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[[[\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^3\ [\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^4]^2]^{C_1}]^0 \\ \xrightarrow{\mathsf{Ret}(0\cdot C_1\cdot 2,3,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[[(\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)\ [\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^4]^2]^{C_1}]^0 \\ \xrightarrow{\mathsf{Ret}(0\cdot C_1\cdot 2,4,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[[(\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)\ (\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)]^2]^{C_1}]^0 \\ \xrightarrow{\beta(0\cdot C_1\cdot 2,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[[[[\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^6]^{C_2}]^2]^{C_1}]^0 \\ \xrightarrow{\mathsf{Ret}(0\cdot C_1\cdot 2\cdot C_2,6,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[[[\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^{C_2}]^2]^{C_1}]^0 \\ \xrightarrow{\mathsf{Ret}(0\cdot C_1\cdot 2,C_2,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[[\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^2]^{C_1}]^0 \\ \xrightarrow{\mathsf{Ret}(0\cdot C_1,2,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [[\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^{C_1}]^0 \\ \xrightarrow{\mathsf{Ret}(0,C_1,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & [\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6]^0 \\ \xrightarrow{\mathsf{Ret}(\varepsilon,0,\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)} & \lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6 \end{array}$$

where the event $C_1 = \mathsf{Call}(\lambda^{\ell_1}\mathrm{x}.\,[[\mathrm{x}]^3\ [\mathrm{x}]^4]^2, \lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)$ and the event $C_2 = \mathsf{Call}(\lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6, \lambda^{\ell_2}\mathrm{y}.\,[\mathrm{y}]^6)$.

Figure 1. Example of reductions in the labelled λ-calculus.

2.3 Trace Collecting Semantics

We use σ to range over value substitutions, *i.e.* finite mappings from variables to *closed* values. The expression $t \cdot \sigma$ denotes the application of the value substitution σ to the term t: it amounts to replacing every free occurrence of x in t with $\sigma(\mathrm{x})$. We write $\sigma[\mathrm{x} \mapsto v]$ for the substitution σ extended with a new binding that maps x to v. Given a set of substitutions $\mathcal{I}$, we write $\mathcal{I}[\mathrm{x} \mapsto v]$ for the set $\{\sigma[\mathrm{x} \mapsto v] \mid \sigma \in \mathcal{I}\}$, and write $\mathcal{I}[\mathrm{x} \mapsto S]$ for the set $\{\sigma[\mathrm{x} \mapsto v] \mid \sigma \in \mathcal{I}, v \in S\}$.

We consider the following semantics for terms that, given a set of input substitutions, produces a set of pairs, composed of the execution trace and the output value.

Definition 2.7. $(\!|t|\!)_{\mathcal{I}} \triangleq \{(\mathrm{tr}, v) \mid t \cdot \sigma \overset{\mathrm{tr}}{\longrightarrow}{}^{*} v,\ \sigma \in \mathcal{I}\}$

Definition 2.8 (Semantic preorder)**.** We write $\mathrm{tr}_1 \leq \mathrm{tr}_2$ when tr_1 is a sub-sequence of tr_2. It is an order relation. We use the same symbol to denote the following pre-order on sets of traces and values:

$$S_1 \leq S_2 \triangleq \forall(\mathrm{tr}_1, v) \in S_1, \exists \mathrm{tr}_2, \mathrm{tr}_1 \leq \mathrm{tr}_2 \wedge (\mathrm{tr}_2, v) \in S_2.$$

The pre-order $\leq$ will serve as the approximation ordering. Larger sets of traces and values are less precise, in the sense that they might contain more traces and output values, and might also contain traces with more events.

Program reduction and trace ordering are related by the following lemma:

Lemma 2.9 (Preservation). *If* $t_1 \xrightarrow{\mathsf{tr}}{}^{*} t_2$, *then* $\llparenthesis t_2 \rrparenthesis_{\mathcal{I}} \leq \llparenthesis t_1 \rrparenthesis_{\mathcal{I}}$.

The above lemma is central for our semantic framework: it implies that any correct over-approximation of the semantics of t_1 (in the sense of $\leq$) is necessarily a sound approximation of any reduct of t_1. In other words, the semantic property of being a sound approximation is preserved by reduction. This fact is crucial for the preservation property of CFA constraints from §4 (Lemma 4.2). The proof of Lemma 2.9 is direct: the output values of t_1 and t_2 are the same, and the trace produced by t_2 is a suffix—and thus a sub-sequence—of the trace of t_1.

In the rest of this paper, we will only consider *fully annotated* terms as our source language, *i.e.* terms where variables, applications and abstractions are enclosed in exactly one annotation, which must be a program point. A fully annotated value is an abstraction of a fully annotated term.

For each abstraction that we define in the rest of the paper, we derive the inclusion properties that are satisfied by the abstract semantics. These inclusions naturally lead to the design of analysers in the style of *big-step* static interpreters [4, 9], that follow the structure of programs.

For fully annotated terms, we get the following inclusions for the trace semantics:

Lemma 2.10 (Semantic inclusions for annotated terms).

$$\llparenthesis [\mathsf{x}]^{p} \rrparenthesis_{\mathcal{I}} = \{(\mathsf{Ret}(\varepsilon, p, \sigma(\mathsf{x})), \sigma(\mathsf{x})) \mid \sigma \in \mathcal{I}\}$$

$$\llparenthesis [\lambda^{\ell}\mathsf{x}.\, t]^{p} \rrparenthesis_{\mathcal{I}} = \{(\mathsf{Ret}(\varepsilon, p, (\lambda^{\ell}\mathsf{x}.\, t)\cdot\sigma), (\lambda^{\ell}\mathsf{x}.\, t)\cdot\sigma) \mid \sigma \in \mathcal{I}\}$$

$$\llparenthesis [[t_1]^{p_1}\ [t_2]^{p_2}]^{p} \rrparenthesis_{\mathcal{I}} \subseteq \left\{ \begin{array}{l|l} (p\cdot\mathsf{tr}_1 \mathbin{+\!\!+} p\cdot\mathsf{tr}_2 \mathbin{+\!\!+} & \\ \beta(p, \lambda^{\ell}\mathsf{x}.\,[t_3]^{p_3}, v_2) \mathbin{+\!\!+} & (\mathsf{tr}_1, \lambda^{\ell}\mathsf{x}.[t_3]^{p_3}) \in \llparenthesis [t_1]^{p_1} \rrparenthesis_{\mathcal{I}} \\ p\cdot\mathsf{Call}(\lambda^{\ell}\mathsf{x}.\,[t_3]^{p_3}, v_2)\cdot\mathsf{tr}_3 \mathbin{+\!\!+} & \wedge\,(\mathsf{tr}_2, v_2) \in \llparenthesis [t_2]^{p_2} \rrparenthesis_{\mathcal{I}} \\ \mathsf{Ret}(p, \mathsf{Call}(\lambda^{\ell}\mathsf{x}.[t_3]^{p_3}, v_2), v_3) & \wedge\,(\mathsf{tr}_3, v_3) \in \llparenthesis [t_3]^{p_3} \rrparenthesis_{\mathcal{I}[\mathsf{x}\mapsto v_2]} \\ \mathbin{+\!\!+}\mathsf{Ret}(\varepsilon, p, v_3), & \\ v_3) & \end{array} \right\}$$

For variables, the semantics produces return events only, where the returned values are read in the possible input substitutions. The case for functions is similar: the returned values are functions whose free variables are substituted for the values given by the input substitutions.

In the inclusion for fully annotated applications, the execution trace tr_3 that corresponds to the function call is *enclosed* in the following two events: the event $\beta(p, \lambda^{\ell}\mathsf{x}.\,[t_3]^{p_3}, v_2)$ that denotes the *start* of the function call, and the event $\mathsf{Ret}(p, \mathsf{Call}(\lambda^{\ell}\mathsf{x}.\,[t_3]^{p_3}, v_2), v_3)$, that denotes the *return* from that function call. Interestingly, the same trace tr_3 is *prefixed* with the location $p\cdot\mathsf{Call}(\lambda^{\ell}\mathsf{x}.\,[t_3]^{p_3}, v_2)$, that records which function call is considered, and where it happened. In usual CFA analyses for functional languages, only the *locus* p of the function call is remembered.

$$\begin{tikzcd}
\mathcal{T} \to \mathbb{R}(\widehat{\mathsf{Clos}}) \times (\mathbb{X} \xrightarrow{\text{fin}} (\mathbb{L}\times\mathbb{D})) \to \mathbb{R}(\widehat{\mathsf{Clos}}) \times \mathbb{C}(\widehat{\mathsf{Clos}}) \arrow[d, shift right, "\gamma_{\mathbb{D}}(\phi)"'] \\
\mathcal{T} \to \mathbb{R}(\widehat{\mathsf{Clos}}) \times (\mathbb{X} \xrightarrow{\text{fin}} (\mathbb{L}\times\mathbb{D})) \to \mathbb{R}(\widehat{\mathsf{Clos}}) \times \mathbb{C}(\widehat{\mathsf{Clos}}) \arrow[u, shift right, "\alpha_{\mathbb{D}}(\phi)"'] \arrow[u, bend right=60, "\text{context projection (§3.7)}"'] \arrow[d, "\gamma_{\hat{\mathbb{G}}}"'] \\
\mathcal{T} \to (\mathbb{X} \xrightarrow{\text{fin}} \wp(\mathsf{Clos})) \to \mathbb{R}(\mathsf{Clos}) \times \mathbb{C}(\mathsf{Clos}) \arrow[u, bend right=60, "\text{indirection abstraction (§3.6)}"'] \arrow[d, "\gamma_{\mathsf{clos}}"'] \\
\mathcal{T} \to (\mathbb{X} \xrightarrow{\text{fin}} \wp(\mathbb{V})) \to \mathbb{R}(\mathbb{V}) \times \mathbb{C}(\mathbb{V}) \arrow[u, bend right=60, "\text{closure abstraction (§3.5)}"'] \arrow[d, shift right, "\gamma_{\mathsf{M}}"'] \\
\mathcal{T} \to \wp(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \to \mathbb{R}(\mathbb{V}) \times \mathbb{C}(\mathbb{V}) \arrow[u, shift right, "\alpha_{\mathsf{M}}"'] \arrow[u, bend right=60, "\text{independent attribute for inputs (§3.4)}"'] \arrow[d, shift right, "\gamma_{\mathsf{EC}}"'] \\
\mathcal{T} \to \wp(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \to \wp(\mathbb{R}(\mathbb{V})) \times \wp(\mathbb{C}(\mathbb{V})) \arrow[u, shift right, "\alpha_{\mathsf{EC}}"'] \arrow[u, bend right=60, "\text{independent attribute for maps (§3.3)}"'] \arrow[d, shift right, "\gamma_{\mathsf{IA}}"'] \\
\mathcal{T} \to \wp(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \to \wp(\mathbb{R}(\mathbb{V}) \times \mathbb{C}(\mathbb{V})) \arrow[u, shift right, "\alpha_{\mathsf{IA}}"'] \arrow[u, bend right=60, "\text{independent attribute for pairs (§3.3)}"'] \arrow[d, shift right, "\gamma_{\mathsf{NoVal}}"'] \\
\mathcal{T} \to \wp(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \to \wp(\mathbb{R}(\mathbb{V}) \times \mathbb{C}(\mathbb{V}) \times \mathbb{V}) \arrow[u, shift right, "\alpha_{\mathsf{NoVal}}"'] \arrow[u, bend right=60, "\text{output oblivion (§3.3)}"'] \arrow[d, shift right, "\gamma_{\mathsf{h}}"'] \\
\mathcal{T} \to \wp(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \to \wp(\mathbb{T} \times \mathbb{V}) \arrow[u, shift right, "\alpha_{\mathsf{h}}"'] \arrow[u, bend right=60, "\text{history oblivion (§3.2)}"'] \arrow[d, shift right, "\gamma_{\mathsf{e}}"'] \\
\mathcal{T} \to \wp(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \to \wp(\mathbb{T} \times \mathbb{V}) \arrow[u, shift right, "\alpha_{\mathsf{e}}"'] \arrow[u, bend right=60, "\text{event selection (§3.1)}"']
\end{tikzcd}$$

Figure 2. Abstraction steps.

3 A Systematic Reconstruction of CFA

Standard control flow analyses for a program t construct two objects as results of the analysis. The first object is a *global environment* $\hat{\rho}$ that records the set of values that variables of t could have been assigned to. To distinguish between several instances of variables, variables are distinguished by their calling context—sometimes also called memento [31, 33] or instances [37]. A global environment maps a pair (x, δ) of a variable and its calling context to a set of abstract values. The global environment is called *global store* in the AAM line of work. In this paper, instead of taking variables and contexts (x, δ) as keys of global environments, we choose *binding sites* (corresponding to λ-abstractions) and contexts (ℓ, δ) as keys. This choice is technical and can be ignored in a first read: it ensures that our definitions do not depend on the *names* of variables, and are consistent with the α-equivalence of λ-terms. We write $\mathbb{R}(A) \triangleq (\mathbb{L}\times\mathbb{D}) \to \wp(A)$ for the type of global environments that contain sets of elements of type A.

The second object computed by a CFA is a *global cache* $\hat{C}$, that maps every sub-term of the program t—identified by their program points—to a set of abstract values that denotes the possible evaluation results of that sub-term. Similarly to global environments, different versions of sub-terms are distinguished by their evaluation contexts. A global cache therefore maps a pair (p, δ) of a program point and an evaluation context to a set of abstract values. Some analyses remove the need of a cache by working with administrative normal forms (ANF), where every sub-term is bound to a variable. We write $\mathbb{C}(A) \triangleq (\mathbb{P} \times \mathbb{D}) \to \wp(A)$ for the type of global caches that contain sets of elements of type A.

In this section, we show how to reconstruct the environments and caches of k-CFA as abstractions of the trace-based semantics. The abstraction strategy is summarised in Fig. 2. Most abstractions form Galois connections [3]. A Galois connection between partially ordered sets is defined as follows:

Definition 3.1 (Galois connection)**.** We write $(A, \leq_A) \leftrightarrows^{\gamma}_{\alpha} (B, \leq_B)$ when for any $a \in A$ and $b \in B$, $\alpha(a) \leq_B b \Leftrightarrow a \leq_A \gamma(b)$.

The first stages of abstraction—starting from the bottom of Fig. 2—are responsible for selecting events in the traces, and forgetting about the ordering of events. After these preliminary steps, we have built environments and caches of types $\mathbb{R}(\mathbb{V})$ and $\mathbb{C}(\mathbb{V})$, that contain sets of values. Then, inputs are abstracted and finally values are abstracted by the closure abstraction and the indirection abstraction. These two abstractions are not Galois connections, as there is no *best* way to perform these abstractions, but they still lead to correct analyses.

The closure abstraction builds abstract closures, composed of some code and some environment, that maps variables to sets of abstract closures. Then, the indirection abstraction introduces intermediate names of type $\mathbb{L} \times \mathbb{D}$: at this level, abstract closures are composed of some code together with an environment that maps variables to names. This indirection is similar to the indirection used in the small-step semantics of the AAM line of work, where names are called *addresses*.

Finally, the last abstraction step modifies the calling contexts in names using a projection function ϕ: we recover k-CFA by choosing $\phi(\delta) = \lfloor \delta \rfloor_k$, that is the truncation to the last k elements of δ.

In the following subsections, we define the abstractions of the semantics and, at some key steps, show how they impact the semantic inclusions. We exploit those inclusions in §4 to semantically justify the constraints of k-CFA.

3.1 Abstraction of Events

We select events by performing two transformations on traces. First, we select the *call sites* in the contexts of events. The call sites are the program points that occur just before a $\mathsf{Call}(v_1, v_2)$ annotation. Recall the example from §2.1 with $[(\lambda^\ell \mathsf{x}.\, t)\, v]^p$ that reduces to $[[t[\mathsf{x} \leftarrow v]]^{\mathsf{Call}(\lambda^\ell \mathsf{x}.\, t, v)}]^p$. Subsequent reductions will happen in a context of the form $\pi = p \cdot \mathsf{Call}(\lambda^\ell \mathsf{x}.\, t, v) \cdot \pi'$, recording that the call site of the actual call is p. Thus, to recover the standard call site information we just need to keep the information $\mathrm{calls}(\pi) = p \cdot \mathrm{calls}(\pi')$. The call filtering is extended to events and traces as follows:

Definition 3.2 (Call filtering)**.**

$$\begin{array}{ll}
\text{On paths:} & \mathrm{calls}\,\varepsilon \triangleq \mathrm{calls}(a \cdot \varepsilon) \triangleq \varepsilon \\
 & \mathrm{calls}(p \cdot \mathsf{Call}(v_1, v_2) \cdot \pi) \triangleq p \cdot \mathrm{calls}\,\pi \\
 & \mathrm{calls}(p_1 \cdot p_2 \cdot \pi) \triangleq \mathrm{calls}(p_2 \cdot \pi) \\
\text{On events:} & \mathrm{calls}(\mathsf{Ret}(\pi, a, v)) \triangleq \mathsf{Ret}(\mathrm{calls}\,\pi, a, v) \\
 & \mathrm{calls}(\beta(\pi, v_1, v_2)) \triangleq \beta(\mathrm{calls}\,\pi, v_1, v_2) \\
\text{On traces:} & \mathrm{calls}\,\varepsilon \triangleq \varepsilon \\
 & \mathrm{calls}(\mathrm{e}, \mathrm{tr}) \triangleq (\mathrm{calls}\,\mathrm{e}), (\mathrm{calls}\,\mathrm{tr})
\end{array}$$

We deliberately drop the Call annotations, because our goal is to recover eventually the *call strings* that are used in k-CFA. We leave to future work the question of keeping Call annotations to obtain other forms of context sensitivity.

The second selection pass on events consists in keeping only events that are labelled by program points—as opposed to call annotations. This is simply done as follows:

Definition 3.3 (Event filtering)**.**

$$\begin{array}{rcl}
\mathrm{ret}\,\varepsilon & \triangleq & \varepsilon \\
\mathrm{ret}(\beta(\pi, v_1, v_2), \mathrm{tr}) & \triangleq & \beta(\pi, v_1, v_2), \mathrm{ret}\,\mathrm{tr} \\
\mathrm{ret}(\mathsf{Ret}(\pi, p, v), \mathrm{tr}) & \triangleq & \mathsf{Ret}(\pi, p, v), \mathrm{ret}\,\mathrm{tr} \\
\mathrm{ret}(\mathsf{Ret}(\pi, \mathsf{Call}(v_1, v_2), v), \mathrm{tr}) & \triangleq & \mathrm{ret}\,\mathrm{tr}
\end{array}$$

By composing the two filter functions, we obtain the abstraction function α_{e} for events, together with its corresponding concretization γ_{e}:

Definition 3.4 (Event abstraction and concretization)**.**

$$\begin{array}{rcl}
\alpha_{\mathrm{e}}(S) & \triangleq & \{(\mathrm{ret} \circ \mathrm{calls}(\mathrm{tr}), v) \mid (\mathrm{tr}, v) \in S\} \\
\gamma_{\mathrm{e}}(S) & \triangleq & \{(\mathrm{tr}, v) \mid \mathrm{ret} \circ \mathrm{calls}(\mathrm{tr}) \leq \mathrm{tr}' \wedge (\mathrm{tr}', v) \in S\}
\end{array}$$

Lemma 3.5. *The functions α_{e} and γ_{e} form a Galois connection: $(\wp(\mathbb{T} \times \mathbb{V}), \leq) \leftrightarrows^{\gamma_{\mathrm{e}}}_{\alpha_{\mathrm{e}}} (\wp(\mathbb{T} \times \mathbb{V}), \leq)$.*

The proof relies on the monotonicity of calls and ret with respect to the sub-sequence order.

The function $\alpha_{\mathrm{e}}\llparenthesis t \rrparenthesis_{\mathcal{I}}$ for a fully annotated term t satisfies the following inclusions:

Lemma 3.6 (Semantic inclusions after event abstraction)**.**

$$\alpha_{\mathrm{e}}\llparenthesis [\mathsf{x}]^p \rrparenthesis_{\mathcal{I}} = \{(\mathsf{Ret}(\varepsilon, p, \sigma(\mathsf{x})), \sigma(\mathsf{x})) \mid \sigma \in \mathcal{I}\}$$

$$\alpha_{\mathrm{e}}\llparenthesis [\lambda^\ell \mathsf{x}.\, t]^p \rrparenthesis_{\mathcal{I}} = \{(\mathsf{Ret}(\varepsilon, p, (\lambda^\ell \mathsf{x}.\, t) \cdot \sigma), (\lambda^\ell \mathsf{x}.\, t) \cdot \sigma) \mid \sigma \in \mathcal{I}\}$$

$$\alpha_{\mathrm{e}}\llparenthesis [[t_1]^{p_1}\ [t_2]^{p_2}]^p \rrparenthesis_{\mathcal{I}} \leq \left\{ \begin{array}{l|l}
(\mathrm{tr}_1 \mathbin{+\!\!+} \mathrm{tr}_2 \mathbin{+\!\!+} & (\mathrm{tr}_1, \lambda^\ell \mathsf{x}.\, [t_3]^{p_3}) \in \alpha_{\mathrm{e}}\llparenthesis [t_1]^{p_1} \rrparenthesis_{\mathcal{I}} \\
\beta(p, \lambda^\ell \mathsf{x}.\, [t_3]^{p_3}, v_2) \mathbin{+\!\!+} & \wedge\, (\mathrm{tr}_2, v_2) \in \alpha_{\mathrm{e}}\llparenthesis [t_2]^{p_2} \rrparenthesis_{\mathcal{I}} \\
p \cdot \mathrm{tr}_3 \mathbin{+\!\!+} \mathsf{Ret}(\varepsilon, p, v_3), & \wedge\, (\mathrm{tr}_3, v_3) \in \alpha_{\mathrm{e}}\llparenthesis [t_3]^{p_3} \rrparenthesis_{\mathcal{I}[\mathsf{x} \mapsto v_2]} \\
v_3) &
\end{array} \right\}$$

Compared to Lemma 2.10, we notice that the events and contexts involved in a call annotation have been removed. In particular, the trace tr_3 that corresponds to the evaluation of the function body obtained from t_1 is now only prefixed by the *call site* p, *i.e.* the program point of the application.

3.2 Unordered History Abstraction

In this abstraction step, we forget the order of events in the computation trace. Instead of a trace, we produce two maps that contain sets of values.

The first map is a global environment that inspects the β-reduction events and remembers on which arguments the functions were called, and in which context. This environment map is defined by the env function.

The second map is a global cache, that reads the return events and remembers which values were produced for the different program points, and in which context. This cache map is defined by the cache function.

Definition 3.7 (Building environments and caches).

$$
\begin{array}{rcl}
\mathrm{env}(\varepsilon) & \triangleq & \dot{\emptyset} \\
\mathrm{env}(\mathrm{Ret}(\delta, p, v), \mathrm{tr}) & \triangleq & \mathrm{env}\,\mathrm{tr} \\
\mathrm{env}(\beta(\delta, \lambda^{\ell}\mathsf{x}.\, t, v), \mathrm{tr}) & \triangleq & \{(\ell, \delta) \mapsto \{v\}\} + \mathrm{env}\,\mathrm{tr} \\
\mathrm{cache}(\varepsilon) & \triangleq & \dot{\emptyset} \\
\mathrm{cache}(\mathrm{Ret}(\delta, p, v), \mathrm{tr}) & \triangleq & \{(p, \delta) \mapsto \{v\}\} + \mathrm{cache}\,\mathrm{tr} \\
\mathrm{cache}(\beta(\delta, v_1, v_2), \mathrm{tr}) & \triangleq & \mathrm{cache}\,\mathrm{tr}
\end{array}
$$

The definitions of env and cache rely on the following operations on maps with a set-valued range.

Definition 3.8 (Operations and relations on maps).

$$
\begin{array}{rcl}
\dot{\emptyset} & \triangleq & \lambda k.\emptyset \\
\{k \mapsto S\} & \triangleq & \lambda k'.\mathrm{if}\ k' = k\ \mathrm{then}\ S\ \mathrm{else}\ \emptyset \\
\pi \cdot m & \triangleq & \lambda(x, \pi').\mathrm{if}\ \pi' = \pi \cdot \pi''\ \mathrm{then}\ m(\pi'')\ \mathrm{else}\ \emptyset \\
m_1 + m_2 & \triangleq & \lambda k.m_1(k) \cup m_2(k) \\
m_1 \mathrel{\dot{\subseteq}} m_2 & \triangleq & \forall k, m_1(k) \subseteq m_2(k)
\end{array}
$$

The above definitions comprise the empty map, the singleton map, how to prefix a map with a path, how to take the union of two maps, and how to compare them. The definitions of history abstraction and concretization follow.

Definition 3.9 (History abstraction and concretization).

$$
\begin{array}{l}
\alpha_{\mathsf{h}}(S) \triangleq \{(\mathrm{env}\,\mathrm{tr}, \mathrm{cache}\,\mathrm{tr}, v) \mid (\mathrm{tr}, v) \in S\} \\
\gamma_{\mathsf{h}}(S) \triangleq \{(\mathrm{tr}, v) \mid \mathrm{env}\,\mathrm{tr} \mathrel{\dot{\subseteq}} \rho \wedge \mathrm{cache}\,\mathrm{tr} \mathrel{\dot{\subseteq}} C \wedge (\rho, C, v) \in S\} \\
S_1 \sqsubseteq S_2 \triangleq \forall (\rho_1, C_1, v_1) \in S_1, \exists (\rho_2, C_2, v_2) \in S_2, \\
\qquad\qquad \rho_1 \mathrel{\dot{\subseteq}} \rho_2 \wedge C_1 \mathrel{\dot{\subseteq}} C_2 \wedge v_1 = v_2
\end{array}
$$

Lemma 3.10. $(\wp(\mathbb{T}\times\mathbb{V}), \leq) \leftrightarrows^{\gamma_{\mathsf{h}}}_{\alpha_{\mathsf{h}}} (\wp(\mathbb{R}(\mathbb{V})\times\mathbb{C}(\mathbb{V})\times\mathbb{V}), \sqsubseteq)$

The proof relies on the monotonicity of env and cache.

We write $A_{\mathsf{h}} = \alpha_{\mathsf{h}} \circ \alpha_{\mathsf{e}}$ for the chaining of the abstractions we have defined so far. The abstract semantics of a fully annotated term $A_{\mathsf{h}}\llparenthesis t \rrparenthesis_{\mathcal{I}}$ satisfies the following inclusions:

Lemma 3.11 (Semantic inclusions after history abstraction).

$$
\begin{array}{l}
A_{\mathsf{h}}\llparenthesis [\mathsf{x}]^{p} \rrparenthesis_{\mathcal{I}} = \{(\dot{\emptyset}, \{(p, \varepsilon) \mapsto \{\sigma(\mathsf{x})\}\}, \sigma(\mathsf{x})) \mid \sigma \in \mathcal{I}\} \\
A_{\mathsf{h}}\llparenthesis [\lambda^{\ell}\mathsf{x}.\, t]^{p} \rrparenthesis_{\mathcal{I}} = \{(\dot{\emptyset}, \{(p, \varepsilon) \mapsto \{v\}\}, v) \mid v = (\lambda^{\ell}\mathsf{x}.\, t)\cdot\sigma, \sigma \in \mathcal{I}\} \\
A_{\mathsf{h}}\llparenthesis [[t_1]^{p_1}\ [t_2]^{p_2}]^{p} \rrparenthesis_{\mathcal{I}} \sqsubseteq \\
\left\{ \begin{array}{l|l}
(\rho_1 + \rho_2 + p \cdot \rho_3 + & \\
\{(\ell, p) \mapsto \{v_2\}\}, & (\rho_1, C_1, \lambda^{\ell}\mathsf{x}.\, [t_3]^{p_3}) \in A_{\mathsf{h}}\llparenthesis [t_1]^{p_1} \rrparenthesis_{\mathcal{I}} \\
C_1 + C_2 + p \cdot C_3 + & \wedge (\rho_2, C_2, v_2) \in A_{\mathsf{h}}\llparenthesis [t_2]^{p_2} \rrparenthesis_{\mathcal{I}} \\
\{(p, \varepsilon)) \mapsto \{v_3\}\}, & \wedge (\rho_3, C_3, v_3) \in A_{\mathsf{h}}\llparenthesis [t_3]^{p_3} \rrparenthesis_{\mathcal{I}[\mathsf{x}\mapsto v_2]} \\
v_3) &
\end{array} \right\}
\end{array}
$$

The call site p prefixes the environment ρ_3 and cache C_3 that result from the evaluation of the function body of the application. Moreover, the global environment records that the arguments of these functions contain the possible values for the arguments of the application. Finally, the global cache remembers that the whole term, *i.e.* the program point p in the empty environment, evaluates to the possible values of the function body.

3.3 Forgetting Output Values and Creation of Environments and Caches

The next abstraction steps are administrative, simple, and use mostly standard Galois connections. We only give a brief sketch of how they work.

First, we only keep the environment and the cache, using the abstraction function $\alpha_{\mathsf{NoVal}}(S) \triangleq \{(\rho, C) \mid \exists v, (\rho, C, v) \in S\}$. Then, we separate environments from caches: using the function $\alpha_{\mathsf{IA}}(S) \triangleq (\{\rho \mid \exists C, (\rho, C) \in S\}, \{C \mid \exists \rho, (\rho, C) \in S\})$, we transform sets of pairs of maps into pairs of sets of maps. Finally each set—of environments or caches—is abstracted as a map—an environment or a cache, respectively—using the function $\alpha_{\mathsf{M}}(S) \triangleq \lambda k. \bigcup_{m\in S} m(k)$ to abstract the maps, and $\alpha_{\mathsf{EC}}(S_1, S_2) \triangleq (\alpha_{\mathsf{M}}(S_1), \alpha_{\mathsf{M}}(S_2))$ to abstract their pairs. All these functions are adjoints of Galois connections.

We define the function $A_{\mathsf{EC}} \triangleq \alpha_{\mathsf{EC}} \circ \alpha_{\mathsf{IA}} \circ \alpha_{\mathsf{NoVal}} \circ A_{\mathsf{h}}$, that stacks up all the abstractions we have defined so far. The semantic inclusions for the abstract semantics of a fully annotated term $A_{\mathsf{EC}}\llparenthesis [t]^{p} \rrparenthesis_{\mathcal{I}}$ are easily obtained, by noticing that for a term $[t]^{p}$ with an output value v and a cache C obtained by $A_{\mathsf{h}}\llparenthesis [t]^{p} \rrparenthesis_{\mathcal{I}}$, we always have $\{v\} \subseteq C(p, \varepsilon)$.

3.4 Abstracting Inputs

The next abstraction abstracts the set of inputs $\mathcal{I}$. We transform this set of substitutions—*i.e.* maps from variables to values—into a map from variables to sets of values, using the Galois connection $\leftrightarrows^{\gamma_{\mathsf{M}}}_{\alpha_{\mathsf{M}}}$ we defined in the previous section. We denote by E the maps from variables to sets of values. The abstract semantics $A_{\mathsf{EC}}\llparenthesis [t]^{p} \rrparenthesis_{\gamma_{\mathsf{M}}(E)}$ of a fully annotated term $[t]^{p}$ satisfies the following inclusions:

Lemma 3.12 (Semantic inclusions after input abstraction).

$$
\begin{array}{l}
A_{\mathsf{EC}}\llparenthesis [\mathsf{x}]^{p} \rrparenthesis_{\gamma_{\mathsf{M}}(E)} \mathrel{\dot{\subseteq}^2} (\dot{\emptyset}, \{(p, \varepsilon) \mapsto E(\mathsf{x})\}) \\
A_{\mathsf{EC}}\llparenthesis [\lambda^{\ell}\mathsf{x}.t]^{p} \rrparenthesis_{\gamma_{\mathsf{M}}(E)} = (\dot{\emptyset}, \{(p, \varepsilon) \mapsto \{(\lambda^{\ell}\mathsf{x}.t) \cdot \sigma \mid \sigma \in \gamma_{\mathsf{M}}(E)\}\}) \\
A_{\mathsf{EC}}\llparenthesis [[t_1]^{p_1}\ [t_2]^{p_2}]^{p} \rrparenthesis_{\gamma_{\mathsf{M}}(E)} \mathrel{\dot{\subseteq}^2} (\rho_1 + \rho_2 + \rho_3, C_1 + C_2 + C_3) \\
\textit{where}\ (\rho_1, C_1) = A_{\mathsf{EC}}\llparenthesis [t_1]^{p_1} \rrparenthesis_{\gamma_{\mathsf{M}}(E)} \\
\textit{and}\ (\rho_2, C_2) = A_{\mathsf{EC}}\llparenthesis [t_2]^{p_2} \rrparenthesis_{\gamma_{\mathsf{M}}(E)} \\
\rho_3 = \sum_{\lambda^{\ell}\mathsf{x}.\, [t_3]^{p_3} \in C_1(p_1, \varepsilon)} p \cdot \mathrm{fst} \circ A_{\mathsf{EC}}\llparenthesis [t_3]^{p_3} \rrparenthesis_{\gamma_{\mathsf{M}}\{\mathsf{x}\mapsto C_2(p_2, \varepsilon)\}} \\
\qquad + \{(\ell, p) \mapsto C_2(p_2, \varepsilon)\} \\
C_3 = \sum_{\lambda^{\ell}\mathsf{x}.\, [t_3]^{p_3} \in C_1(p_1, \varepsilon)} \\
\qquad \textit{let}\ C_3' = \mathrm{snd} \circ A_{\mathsf{EC}}\llparenthesis [t_3]^{p_3} \rrparenthesis_{\gamma_{\mathsf{M}}\{\mathsf{x}\mapsto C_2(p_2, \varepsilon)\}}\ \textit{in} \\
\qquad p \cdot C_3' + \{(p, \varepsilon) \mapsto C_3'(p_3, \varepsilon)\}
\end{array}
$$

In the case of variables, the cache is built by reading the set of possible values in the environment.

For λ-abstractions, the cache uses the set of values that results from closing the λ-abstraction being analysed under all the possible substitutions that belong to the concretized environment. In other words, it is a closure with a set of possible closing environments. We abstract these sets of closures in the next section.

For applications, the environments and caches are built by combining the environments and caches that are computed for each sub-term of the application with the ones that are obtained for the functions $\lambda^\ell \mathsf{x}.\ [t_3]^{p_3}$ that may be called. The environment ρ_3 records which arguments were given to the possibly called functions, whereas the cache C_3 reports that the result of the whole application contains the results of the execution of every executed function body. Interestingly, the terms $[t_3]^{p_3}$ are analysed in a context that does not involve the initial context $\gamma_{\mathsf{M}}(E)$, because $\lambda^\ell \mathsf{x}.\ [t_3]^{p_3}$ is a *closed* value, and therefore the body $[t_3]^{p_3}$ needs no other input than one for its argument x. For this argument x, we provide the possible values for the arguments of the call.

3.5 Closure Abstraction

In this abstraction step, we introduce an abstract domain to represent sets of closures. An abstract closure is composed of a value—a λ-abstraction—and an environment that assigns to each of the free variables of the value a set of abstract closures. The definition of this abstract domain is *recursive*. This abstract domain will be the starting point of the new analysis ∇CFA we describe in §5.

Definition 3.13 (Abstract closures and their concretization).

$$\mathsf{Clos} \triangleq \mathbb{V} \times \mathbb{G} \qquad \mathbb{G} \triangleq \mathbb{X} \xrightarrow{\text{fin}} \wp(\mathsf{Clos})$$
$$\gamma_{\mathbb{G}} \triangleq \gamma_{\mathsf{M}} \circ \gamma_{\mathsf{clos}} \qquad \gamma_{\mathsf{clos}} = \mathsf{map}_{\gamma'_{\mathsf{clos}}}$$
$$\gamma'_{\mathsf{clos}}(S) \triangleq \bigcup_{\langle \lambda^\ell \mathsf{x}.\, t, \Gamma\rangle \in S} \{(\lambda^\ell \mathsf{x}.\, t)\cdot\sigma \mid \sigma \in \gamma_{\mathbb{G}}(\Gamma))\}$$

The concretization of a set of closures builds the set of values for each closure in the set by closing over all the possible environments, and then takes the union of those sets. The inclusions for the abstracted semantics follow:

Lemma 3.14 (Semantic inclusions after closure abstraction).

$A_{\mathsf{EC}}\llparenthesis [\mathsf{x}]^p \rrparenthesis_{\gamma_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}}(\dot{\emptyset}, \{(p,\varepsilon) \mapsto \Gamma(\mathsf{x})\})$

$A_{\mathsf{EC}}\llparenthesis [\lambda^\ell \mathsf{x}.t]^p \rrparenthesis_{\gamma_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}}(\dot{\emptyset}, \{(p,\varepsilon) \mapsto \{\langle \lambda^\ell \mathsf{x}.t, \Gamma|_{\mathsf{fv}(\lambda^\ell \mathsf{x}.\, t)}\rangle\}\})$

$A_{\mathsf{EC}}\llparenthesis [[t_1]^{p_1} [t_2]^{p_2}]^p \rrparenthesis_{\gamma_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}}(\rho_1 + \rho_2 + \rho'_3, C_1 + C_2 + C'_3)$
where $A_{\mathsf{EC}}\llparenthesis [t_1]^{p_1} \rrparenthesis_{\gamma_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}}(\rho_1, C_1)$
and $A_{\mathsf{EC}}\llparenthesis [t_2]^{p_2} \rrparenthesis_{\gamma_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}}(\rho_2, C_2)$
$\rho'_3 = \sum_{c = \langle \lambda^\ell \mathsf{x}.[t_3]^{p_3}, \Gamma_3\rangle \in C_1(p_1,\varepsilon)} p\cdot\rho_3(c) + \{(\ell, p) \mapsto C_2(p_2,\varepsilon)\}$
$C'_3 = \sum_{c = \langle \lambda^\ell \mathsf{x}.[t_3]^{p_3}, \Gamma_3\rangle \in C_1(p_1,\varepsilon)} p\cdot C_3(c) + \{(p,\varepsilon) \mapsto C_3(c)(p_3,\varepsilon)\}$
where $\forall c = \langle \lambda^\ell \mathsf{x}.\ [t_3]^{p_3}, \Gamma_3\rangle \in C_1(p_1,\varepsilon)$,
$A_{\mathsf{EC}}\llparenthesis [t_3]^{p_3} \rrparenthesis_{\gamma_{\mathbb{G}}(\Gamma_3, \mathsf{x}:C_2(p_2,\varepsilon))} \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}}(\rho_3(c), C_3(c))$

There are two main differences compared to the inclusions of Lemma 3.12. First, in the case of λ-abstractions, an abstract closure is now built, using the input environment to form the closure. Second, in the case of applications, we obtain abstract closures $\langle \lambda^\ell \mathsf{x}.\ [t_3]^{p_3}, \Gamma_3\rangle$ by analysing $[t_1]^{p_1}$. The bodies $[t_3]^{p_3}$ of these closures are then analysed in the environment Γ_3 extended by mapping x to the values from $C_2(p_2, \varepsilon)$, *i.e.* the possible values of the argument t_2.

3.6 Indirection Abstraction

The indirection abstraction is the most complex of this paper, and constitutes a crucial step to eventually recover the textbook definition of k-CFA. The idea of the abstraction is to introduce an indirection through *names* to break the recursion cycle in the definition of closures. Here, we choose *call strings* as names. Previous work from the AAM family have developed this idea too, also to break circularity. They baked the indirection in their semantics using a global store.

The new abstract closures $\hat{c} \in \widehat{\mathsf{Clos}}$ are composed of a λ-abstraction and an environment $\hat{\Gamma} \in \hat{\mathbb{G}}$ that assigns to each free variable a binding label ℓ and a *name* δ—as opposed to a set of closures as done in §3.5. The binding label refers to the binding site of the variable, and the name δ refers to the context in which the variable has been assigned to a value.

Definition 3.15 (Indirection abstraction).

$$\widehat{\mathsf{Clos}} \triangleq \mathbb{V} \times \hat{\mathbb{G}} \qquad \hat{\mathbb{G}} \triangleq \mathbb{X} \xrightarrow{\text{fin}} (\mathbb{L} \times \mathbb{D})$$
$$\langle \lambda^\ell \mathsf{x}.\, t, \Gamma\rangle \mathrel{\in\!\!\!\!+}^{\hat\rho}_{\mathsf{Ind}} S \text{ iff } \exists \hat{\Gamma}, \langle \lambda^\ell \mathsf{x}.\, t, \hat{\Gamma}\rangle \in S \wedge \mathrm{dom}\,\Gamma = \mathrm{dom}\,\hat{\Gamma} \wedge$$
$$\forall \mathsf{x} \in \mathrm{dom}\,\Gamma, \forall c \in \Gamma(\mathsf{x}), c \mathrel{\in\!\!\!\!+}^{\hat\rho}_{\mathsf{Ind}} \hat\rho(\hat\Gamma(\mathsf{x}))$$
$$\gamma^{\hat\rho}_{\mathsf{Ind}}{}'(S) \triangleq \{c \mid c \mathrel{\in\!\!\!\!+}^{\hat\rho}_{\mathsf{Ind}} S\} \qquad \gamma^{\hat\rho}_{\mathsf{Ind}}(m) \triangleq \mathsf{map}_{\gamma^{\hat\rho}_{\mathsf{Ind}}{}'}\, m$$
$$\gamma^{\hat\rho}_{\mathsf{cInd}} \triangleq \gamma_{\mathsf{clos}} \circ \gamma^{\hat\rho}_{\mathsf{Ind}} \qquad \gamma_{\hat{\mathbb{G}}}(\hat\rho; \hat\Gamma) \triangleq \gamma_{\mathbb{G}} \circ \gamma^{\hat\rho}_{\mathsf{Ind}}(\hat\Gamma)$$

The meaning of an abstract closure depends on a global environment $\hat\rho \in \mathbb{R}(\widehat{\mathsf{Clos}})$, that maps pairs (ℓ, δ) to sets of abstract closures with indirection. An abstract closure with indirection denotes the set of the abstract closures that are obtained by unfolding the definitions contained in $\hat\rho$ a *finite number of times.* We formally define this in Def. 3.15, using the inductively defined relation $\mathrel{\in\!\!\!\!+}^{\hat\rho}_{\mathsf{Ind}}$ that performs these unfoldings. Our concretization function $\gamma^{\hat\rho}_{\mathsf{Ind}}{}'$ transforms a set of abstract closures in $\widehat{\mathsf{Clos}}$ into a set of abstract closures in Clos from §3.5. Our definitions are inspired from the *correctness relation* $\mathcal{R}$ of [33, p. 159]—an inductive relation that describes the concrete closures that are obtained by finitely unfolding indirections.

The semantic inclusions that result from the indirection abstraction are similar to those of Lemma 3.14.

Lemma 3.16 (Inclusions after indirection abstraction).

$A_{\mathsf{EC}}\llparenthesis [\mathsf{x}]^p \rrparenthesis_{\gamma_{\hat{\mathbb{G}}}(\hat\rho;\hat\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma^{\hat\rho}_{\mathsf{cInd}}(\dot{\emptyset}, \{(p,\varepsilon) \mapsto \hat\rho(\hat\Gamma(\mathsf{x}))\})$

$A_{\mathsf{EC}}\llparenthesis [\lambda^\ell \mathsf{x}.t]^p \rrparenthesis_{\gamma_{\hat{\mathbb{G}}}(\hat\rho;\hat\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma^{\hat\rho}_{\mathsf{cInd}}(\dot{\emptyset}, \{(p,\varepsilon) \mapsto \{\langle \lambda^\ell \mathsf{x}.t, \hat\Gamma|_{\mathsf{fv}(\lambda^\ell \mathsf{x}.t)}\rangle\}\})$

$A_{\mathsf{EC}}\llparenthesis [[t_1]^{p_1} [t_2]^{p_2}]^p \rrparenthesis_{\gamma_{\hat{\mathbb{G}}}(\hat\rho;\hat\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma^{\hat\rho}_{\mathsf{cInd}}(\hat\rho_1 + \hat\rho_2 + \hat\rho'_3, \hat{C}_1 + \hat{C}_2 + \hat{C}'_3)$
where $A_{\mathsf{EC}}\llparenthesis [t_1]^{p_1} \rrparenthesis_{\gamma_{\hat{\mathbb{G}}}(\hat\rho;\hat\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma^{\hat\rho}_{\mathsf{cInd}}(\hat\rho_1, \hat{C}_1)$
and $A_{\mathsf{EC}}\llparenthesis [t_2]^{p_2} \rrparenthesis_{\gamma_{\hat{\mathbb{G}}}(\hat\rho;\hat\Gamma)} \mathrel{\dot{\subseteq}^2} \gamma^{\hat\rho}_{\mathsf{cInd}}(\hat\rho_2, \hat{C}_2)$
$\hat\rho'_3 = \sum_{\hat{c} = \langle \lambda^\ell \mathsf{x}.[t_3]^{p_3}, \hat\Gamma_3\rangle \in \hat{C}_1(p_1,\varepsilon)} p\cdot\hat\rho_3(\hat{c}) + \{(\ell, p) \mapsto \hat{C}_2(p_2,\varepsilon)\}$
$\hat{C}'_3 = \sum_{\hat{c} = \langle \lambda^\ell \mathsf{x}.[t_3]^{p_3}, \hat\Gamma_3\rangle \in \hat{C}_1(p_1,\varepsilon)} p\cdot\hat{C}_3(\hat{c}) + \{(p,\varepsilon) \mapsto \hat{C}_3(\hat{c})(p_3,\varepsilon)\}$
where $\forall \hat{c} = \langle \lambda^\ell \mathsf{x}.\ [t_3]^{p_3}, \hat\Gamma_3\rangle \in \hat{C}_1(p_1,\varepsilon)$.
$A_{\mathsf{EC}}\llparenthesis [t_3]^{p_3} \rrparenthesis_{\gamma_{\hat{\mathbb{G}}}(\hat\rho;\hat\Gamma_3, \mathsf{x}:(\ell,\delta(\hat{c})))} \mathrel{\dot{\subseteq}^2} \gamma^{\hat\rho}_{\mathsf{cInd}}(\hat\rho_3(\hat{c}), \hat{C}_3(\hat{c}))$
and $\delta(\hat{c})$ *is chosen such that* $\hat{C}_2(p_2,\varepsilon) \subseteq \hat\rho(\ell, \delta(\hat{c}))$

The semantic inclusions exhibit one degree of freedom, in the case of applications. The name $\delta(\hat{c})$ to assign to the arguments of function bodies can be chosen freely, as long as the global environment satisfies that the possible values for the argument in the context $\delta(\hat{c})$ contains the possible values of t_2. This degree of freedom has been thoroughly explored by the AAM line of work, through the use of specific *allocation* functions. In CFA, the standard choice is to take the current enclosing context extended with p (the call site).

In §4, we will exploit the inclusions of Lemma 3.16 to justify the constraint rules for ∞-CFA.

3.7 Call Sites Abstraction: Context Projection

The final abstraction step is to lose information by identifying certain contexts. This is where standard CFAs transform the search space into a finite one. We model this *collapse* of the search space by a projection function $\phi \in \mathbb{D} \to \mathbb{D}$.

Definition 3.17 (Context projection abstraction).

$$\begin{array}{rcl} \alpha_{\mathbb{D}}(\phi)(m) & \triangleq & \lambda(x,\delta).\bigcup_{\{\delta' \mid \phi(\delta')=\delta\}} m(x,\delta') \\ \gamma_{\mathbb{D}}(\phi)(m) & \triangleq & \lambda(x,\delta).\, m(x,\phi(\delta)) \end{array}$$

The action of ϕ on maps is to unite the sets that are assigned to the keys k_1 and k_2, when $\phi(k_1) = \phi(k_2)$. So far, no hypothesis on ϕ is required. The functions ϕ define the *context sensitivity policies* of the analysis. By taking $\phi(\delta) = \delta$, we keep the same abstract semantics, *i.e.* ∞-CFA. If we take $\phi(\delta) = \varepsilon$, there is no distinction between contexts anymore, and we obtain 0-CFA. To get k-CFA, we take $\phi(\delta) = \lfloor\delta\rfloor_k$, *i.e.* the function that keeps the last k elements of δ.

Lemma 3.18. *$\alpha_{\mathbb{D}}(\phi)$ and $\gamma_{\mathbb{D}}(\phi)$ form a Galois connection:* $(\mathbb{R}(\widehat{\mathsf{Clos}}) \times \mathbb{C}(\widehat{\mathsf{Clos}}), \dot{\subseteq}^2) \leftrightarrows^{\gamma_{\mathbb{D}}(\phi)}_{\alpha_{\mathbb{D}}(\phi)} (\mathbb{R}(\widehat{\mathsf{Clos}}) \times \mathbb{C}(\widehat{\mathsf{Clos}}), \dot{\subseteq}^2)$

4 Constraint-Based CFA

As a first illustration of the use of the abstractions defined in the previous sections we show how they enable a semantic and modular soundness proof of the constraint rules for infinitary CFA—where no approximation of contexts is performed—and for ϕ-CFA, a family of analyses parameterised by a projection function ϕ, that generalises uniform k-CFA. More specifically, we give a *semantic* definition of what it means for a pair $(\hat{\rho}, \hat{C})$ to be a solution of a CFA problem. As a consequence, the syntactic constraint rules become *theorems*. A benefit of this approach is that the proof method is *modular*: each rule is proved sound separately, instead of considering the set of rules as a whole.

We use the same notation $(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [t]^p$ as in [33, p. 190], that defines the valid solutions of a CFA problem. This judgement denotes that $(\hat{\rho}, \hat{C})$ is a valid solution for the CFA of the term $[t]^p$, taken in the calling context δ and in the environment $\hat{\Gamma}$. Our semantic definition follows:

Definition 4.1 (Semantic validity of an ∞-CFA solution).

$$(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [t]^p \quad \triangleq \quad \delta \cdot A_{\mathsf{EC}}(\!|[t]^p|\!)_{\gamma_{\hat{\mathbb{G}}}(\hat{\rho},\hat{\Gamma})} \;\dot{\subseteq}^2\; \gamma_{\mathsf{clos}} \circ \gamma^{\hat{\rho}}_{\mathsf{Ind}}(\hat{\rho}, \hat{C})$$

The validity of the solution $(\hat{\rho}, \hat{C})$ is stated as an over-approximation property. Consider the set of inputs $\mathcal{I}$ defined by $\hat{\Gamma}$ under the definitions of $\hat{\rho}$, *i.e.* $\mathcal{I} = \gamma_{\hat{\mathbb{G}}}(\hat{\rho}, \hat{\Gamma})$. The definition tells us that $(\hat{\rho}, \hat{C})$ must be an over-approximation of the semantics of the program $[t]^p$ for the set of inputs $\mathcal{I}$, where this semantics is considered "in the calling context δ". This is why the maps obtained by abstracting the semantics are prefixed with the call site path δ.

From Lemma 2.9 and the monotonicity of the abstractions it follows that semantic validity is preserved by reduction:

Lemma 4.2. *If $t_1 \overset{\mathrm{tr}}{\longrightarrow}{}^{*} t_2$ and $(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} t_1$, then $(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} t_2$.*

The proof method we employ for the preservation property radically differs from previous proofs used in constraint-based approaches [16, 17, 33, 46], where preservation is proved in a *syntactic* manner, by reasoning *globally* on the whole set of rules that define the constraints: their proofs inspect complete derivation trees built from the rules for the initial program, and reconstruct derivation trees for the reduced program. Such proofs follow the style of *subject reduction* proofs for type systems [35], and are hard to maintain and to extend. In contrast, our proof of preservation is short and simple, and is independent of the set of rules, as it relies solely on the definition of semantic validity and on the monotonicity of the abstractions.

Our central result is the following theorem, that asserts the soundness of the rules for ∞-CFA.

Theorem 4.3 (Constraints for ∞-CFA). *The following inference rules are sound:*

$$\frac{\hat{\rho}(\hat{\Gamma}(\mathsf{x})) \subseteq \hat{C}(p,\delta)}{(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [\mathsf{x}]^p} \qquad \frac{\{\langle \lambda^{\ell}\mathsf{x}.\, t, \hat{\Gamma}|_{\mathsf{fv}(\lambda^{\ell}\mathsf{x}.\, t)}\rangle\} \subseteq \hat{C}(p,\delta)}{(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [\lambda^{\ell}\mathsf{x}.\, t]^p}$$

$$\frac{\begin{array}{l} \qquad (\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [t_1]^{p_1} \qquad (\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [t_2]^{p_2} \\ \forall \langle \lambda^{\ell}\mathsf{x}.\, [t_3]^{p_3}, \hat{\Gamma}_3\rangle \in \hat{C}_1(p_1,\delta), \\ \qquad (\hat{\rho}, \hat{C}) \models^{\hat{\Gamma},\mathsf{x}:(\ell,\delta p)}_{\delta p} [t_3]^{p_3} \\ \qquad \wedge\, \hat{C}(p_2,\delta) \subseteq \hat{\rho}(\ell,\delta p) \wedge \hat{C}(p_3,\delta p) \subseteq \hat{C}(p,\delta) \end{array}}{(\hat{\rho}, \hat{C}) \models^{\hat{\Gamma}}_{\delta} [[t_1]^{p_1}\ [t_2]^{p_2}]^p}$$

The proof relies on the inclusions provided by Lemma 3.16. The main arguments to conduct the proof are the monotonicity of the path-prefixing operator, and the fact that both path-prefixing and map-union semi-commute with the concretizations γ_{clos} and γ_{Ind} (*i.e.*, $\delta \cdot \gamma(m) \le \gamma(\delta \cdot m)$ and $\gamma(m_1) + \gamma(m_2) \le \gamma(m_1 + m_2)$ for the order relations of interest). Importantly, each rule is proved sound *independently* of each other. We believe this makes sound control-flow analyses easier to design, and also more robust to extend with new features. To our knowledge, this is the first compositional explanation of a context-sensitive CFA.

By proceeding in a similar way as for ∞-CFA, we can generalise the previous results to the validity of ϕ-CFA, *i.e.* a CFA problem where the calling contexts are projected by a function ϕ. The statement differs from Def. 4.1 only by adding a call to the abstraction function $\alpha_{\mathbb{D}}(\phi)$ from §3.6.

Definition 4.4 (Semantic validity of a ϕ-CFA solution)**.**

$$(\phi, \hat{\rho}, \hat{C}) \models_{\delta}^{\hat{\Gamma}} [t]^p \quad \triangleq$$
$$\alpha_{\mathbb{D}}(\phi)(\delta \cdot A_{\mathsf{EC}}(\!|[t]^p|\!)_{\gamma_{\hat{\mathbb{G}}}(\hat{\rho},\hat{\Gamma})}) \mathrel{\dot{\subseteq}^2} \gamma_{\mathsf{clos}} \circ \gamma_{\mathsf{Ind}}^{\hat{\rho}}(\hat{\rho}, \hat{C})$$

The following theorem follows. If we take $\phi(\delta) = \lfloor \delta \rfloor_k$, the theorem gives a semantic justification for the textbook constraint rules of uniform k-CFA.

Theorem 4.5 (Constraints for ϕ-CFA)**.** *Assume that $\phi \circ \phi = \phi$ and that δ and every path that occurs in $\hat{\rho}, \hat{C}, \hat{\Gamma}$ are fixpoints of ϕ. Then, the following inference rules are sound:*

$$\frac{\hat{\rho}(\hat{\Gamma}(\mathsf{x})) \subseteq \hat{C}(p, \delta)}{(\phi, \hat{\rho}, \hat{C}) \models_{\delta}^{\hat{\Gamma}} [\mathsf{x}]^p} \qquad \frac{\{\langle \lambda^{\ell}\mathsf{x}.\, t, \hat{\Gamma}|_{\mathrm{fv}(\lambda^{\ell}\mathsf{x}.\, t)} \rangle\} \subseteq \hat{C}(p, \delta)}{(\phi, \hat{\rho}, \hat{C}) \models_{\delta}^{\hat{\Gamma}} [\lambda^{\ell}\mathsf{x}.\, t]^p}$$

$$\frac{\begin{array}{l} \qquad (\hat{\rho}, \hat{C}) \models_{\delta}^{\hat{\Gamma}} [t_1]^{p_1} \qquad (\hat{\rho}, \hat{C}) \models_{\delta}^{\hat{\Gamma}} [t_2]^{p_2} \\ \forall \langle \lambda^{\ell}\mathsf{x}.\, [t_3]^{p_3}, \hat{\Gamma}_3 \rangle \in \hat{C}_1(p_1, \delta), \\ \quad (\hat{\rho}, \hat{C}) \models_{\phi(\delta p)}^{\hat{\Gamma}, \mathsf{x}:(\ell, \phi(\delta p))} [t_3]^{p_3} \\ \quad \wedge \hat{C}(p_2, \delta) \subseteq \hat{\rho}(\ell, \phi(\delta p)) \wedge \hat{C}(p_3, \phi(\delta p)) \subseteq \hat{C}(p, \delta) \end{array}}{(\phi, \hat{\rho}, \hat{C}) \models_{\delta}^{\hat{\Gamma}} [[t_1]^{p_1}\ [t_2]^{p_2}]^p}$$

As expected, the rules differ from those of ∞-CFA only in the application rule, where the projection ϕ is used to project the extended calling context δp during the analysis of the function bodies. The idempotency hypothesis $\phi \circ \phi = \phi$ ensures that the new paths $\phi(\delta p)$ are fixpoints of ϕ, which is made necessary by our requirement that $\hat{\rho}$ and $\hat{C}$ should only contain paths that are fixpoints of ϕ.

In k-CFA, the chosen ϕ is such that ϕ has a *finite range*. This ensures that there is a finite number of $\hat{\Gamma}$, $\hat{\rho}$ and $\hat{C}$. Consequently, the resolution of the system of constraints necessarily explores a finite number of states, because the set of states has been *made* finite. In §5, we consider instead a system where the set of states is infinite, but where the algorithm is guaranteed to explore a finite number of them. This is achieved by widening techniques.

To conclude this section, we give the precise semantic property that a solution to ϕ-CFA enjoys.

Theorem 4.6. *Assume $(\phi, \hat{\rho}, \hat{C}) \models_{\varepsilon}^{\{\,\}} [t_0]^{p_0}$ and $[t_0]^{p_0} \xrightarrow{\mathrm{tr}}{}^{*} v_0$ and t_0 is a closed program. Then:*

- *If $\beta(\pi, \lambda^{\ell}\mathsf{x}.\, t, v) \in \mathrm{tr}$, then there exists an abstract closure $\hat{c} \in \hat{\rho}(\ell, \phi(\mathrm{calls}(\pi)))$ such that $v \in \gamma_{\mathsf{cInd}}^{\hat{\rho}}\{\hat{c}\}$.*
- *If $\mathrm{Ret}(\pi, p, v) \in \mathrm{tr}$, then there exists an abstract closure $\hat{c} \in \hat{C}(p, \phi(\mathrm{calls}(\pi)))$ such that $v \in \gamma_{\mathsf{cInd}}^{\hat{\rho}}\{\hat{c}\}$.*

The theorem highlights the meaning of $\hat{\rho}$ and $\hat{C}$ as solutions of the ϕ-CFA analysis of the closed program t_0 considered in the empty context. First, the theorem tells us that for every β-reduction event in the trace of the analysed program, there is a corresponding entry in the global environment $\hat{\rho}$ that contains an abstract value $\hat{c}$ that contains the value used as argument during that β-reduction. Second, for every *return event* in the trace of the program, there is in the cache $\hat{C}$ a corresponding entry that contains an abstract value $\hat{c}$ that contains the returned value as reported by the event.

The proof of Theorem 4.6 proceeds by unfolding Definition 4.4 and exploiting the order relations and the concretization functions of the Galois connections.

5 Beyond the Finite State Abstraction

In this section, we deviate from the finite state approximation methodology usually applied in CFAs, and introduce an abstract domain for closures, that is recursively defined, and whose values can have unbounded depths. To ensure the convergence of the analysis, we use a widening operator, as prescribed by the abstract interpretation theory, that makes ascending chains stationary. We obtain the ∇CFA analysis.

5.1 Abstract Closures with Widening

The abstract domain is derived from the domain of Def. 3.13. The environments in that previous domain could contain infinite sets of closures. In this new domain, we only consider finite sets of closures, and moreover, we enforce that there is at most one abstract environment per function. The abstract closures $c^{\sharp} \in \mathsf{Clos}^{\sharp}$ are thus either a finite map from functions to their closing abstract environment, or the element $\top$, that represents the set of all closures from §3.5.

Definition 5.1 (Abstract Closures)**.**

$$\mathsf{Clos}^{\sharp} \triangleq (\mathbb{V} \xrightarrow{\mathrm{fin}} \mathbb{G}^{\sharp}) + \{\top\} \qquad \mathbb{G}^{\sharp} \triangleq \mathbb{X} \xrightarrow{\mathrm{fin}} \mathsf{Clos}^{\sharp}$$
$$\gamma_{\mathsf{clos}}^{\sharp}(\top) \triangleq \mathsf{Clos} \quad \gamma_{\mathsf{clos}}^{\sharp}(m) \triangleq \textstyle\bigcup_{\langle v, \Gamma^{\sharp} \rangle \in m} \{\langle v, \mathrm{map}_{\gamma_{\mathsf{clos}}^{\sharp}}(\Gamma^{\sharp}) \rangle\}$$

We define the union of abstract closures as follows:

Definition 5.2 (Union of closures)**.**

$$\top \cup_{\mathsf{clos}}^{\sharp} \top \triangleq m \cup_{\mathsf{clos}}^{\sharp} \top \triangleq \top \cup_{\mathsf{clos}}^{\sharp} m \triangleq \top$$
$$m_1 \cup_{\mathsf{clos}}^{\sharp} m_2 \triangleq \lambda v. \begin{cases} m_1(v) \cup_{\mathbb{G}}^{\sharp} m_2(v) & \text{if } v \in \mathrm{dom}\, m_1 \cap \mathrm{dom}\, m_2 \\ m_1(v) & \text{if } v \in \mathrm{dom}\, m_1 \setminus \mathrm{dom}\, m_2 \\ m_2(v) & \text{if } v \in \mathrm{dom}\, m_2 \setminus \mathrm{dom}\, m_1 \end{cases}$$
$$\Gamma_1^{\sharp} \cup_{\mathbb{G}}^{\sharp} \Gamma_2^{\sharp} \triangleq \lambda \mathsf{x}. \Gamma_1^{\sharp}(\mathsf{x}) \cup_{\mathsf{clos}}^{\sharp} \Gamma_2^{\sharp}(\mathsf{x})$$

The interesting part of the definition is for the finite maps, that represent the set of closures. The union is defined as the pointwise union of these maps.

Using solely this abstract union operator in an abstract interpreter could make the analysis diverge, because unions can build infinite ascending chains of values. Therefore, we introduce a widening operator $\nabla_{\mathsf{clos}}^{\sharp}$, that is more approximate than $\cup_{\mathsf{clos}}^{\sharp}$, and ensures that sequences of abstract values built from widenings are eventually stationary. The widening operator is defined as follows:

$$\begin{tikzcd}
\mathcal{T} \to (\mathbb{X} \xrightarrow{\text{fin}} \wp(\mathsf{Clos}^\sharp)) \to \widehat{\mathsf{Clos}} \arrow[d, shift right, "\gamma_{\mathsf{Cache}}(p)"'] \\
\mathcal{T} \to (\mathbb{X} \xrightarrow{\text{fin}} \wp(\mathsf{Clos}^\sharp)) \to \mathbb{C}(\widehat{\mathsf{Clos}}) \arrow[u, shift right, "\alpha_{\mathsf{Cache}}(p)"'] \arrow[u, bend right=60, "\text{cache projection}"'] \arrow[d, shift right, "\gamma_{\mathsf{NoEnv}}"'] \\
\begin{array}{l} \mathcal{T} \to (\mathbb{X} \xrightarrow{\text{fin}} \wp(\mathsf{Clos}^\sharp)) \to \\ \quad ((\mathbb{L} \times \mathbb{D}) \to \mathsf{Clos}^\sharp) \times ((\mathbb{P} \times \mathbb{D}) \to \mathsf{Clos}^\sharp) \end{array} \arrow[u, shift right, "\alpha_{\mathsf{NoEnv}}"'] \arrow[u, bend right=60, "\text{global environment removal}"'] \arrow[d, shift right, "\gamma^\sharp_{\mathsf{clos}}"'] \\
\mathcal{T} \to (\mathbb{X} \xrightarrow{\text{fin}} \wp(\mathsf{Clos})) \to \mathbb{R}(\mathsf{Clos}) \times \mathbb{C}(\mathsf{Clos}) \arrow[u, shift right, "\alpha^\sharp_{\mathsf{clos}}"'] \arrow[u, bend right=60, "\text{finite closure abstraction}"']
\end{tikzcd}$$

Figure 3. Abstraction steps to obtain ∇CFA.

Definition 5.3 (Widening of closures).

$$\top \nabla^\sharp_{\mathsf{clos}} \top \triangleq m \nabla^\sharp_{\mathsf{clos}} \top \triangleq \top \nabla^\sharp_{\mathsf{clos}} m \triangleq \top$$

$$m_1 \nabla^\sharp_{\mathsf{clos}} m_2 \triangleq \lambda v. \begin{cases} m_1(v) \nabla^\sharp_{\mathbb{G}} m_2(v) & \text{if } v \in \operatorname{dom} m_1 \cap \operatorname{dom} m_2 \\ m_1(v) & \text{if } v \in \operatorname{dom} m_1 \setminus \operatorname{dom} m_2 \\ \lfloor m_2(v) \rfloor_{\mathrm{h}(m_1)} & \text{if } v \in \operatorname{dom} m_2 \setminus \operatorname{dom} m_1 \end{cases}$$

$$\Gamma^\sharp_1 \nabla^\sharp_{\mathbb{G}} \Gamma^\sharp_2 \triangleq \lambda \mathsf{x}. \Gamma^\sharp_1(\mathsf{x}) \nabla^\sharp_{\mathsf{clos}} \Gamma^\sharp_2(\mathsf{x})$$

The above definition makes use of $\mathrm{h}(m)$, the height of the finite map m, *i.e.* the maximum of the heights of the elements in the range of m. The definition also involves the function $\lfloor \Gamma^\sharp \rfloor_n$, that truncates each element in the range of $\Gamma^\sharp$ to the height n, *i.e.* replaces sub-structures with $\top$, once the depth n is reached in the structure. The interesting case of the definition is for maps: to compute $m_1 \nabla^\sharp_{\mathsf{clos}} m_2$, the bindings in m_2 that are not already present in m_1 are truncated to the height of m_1. Although the map $m_1 \nabla^\sharp_{\mathsf{clos}} m_2$ can be wider than m_1, its height cannot be larger than m_1's. Because the possible keys of these maps are in finite number (the keys are the λs occurring in the program), the widening chains are eventually stationary.

5.2 Abstractions Towards ∇CFA

The different abstraction steps that lead to ∇CFA are summarised in Fig. 3. The starting point is the abstract semantics we obtained after closure abstraction (§3.5). From there, we apply our new abstraction, that builds the new abstract values of §5.1. Then, we notice in the inclusions of Lemma 3.14 that the global environment ρ is not necessary to compute the cache. So, we remove the global environment ρ, using a standard abstraction. Finally, we choose to only keep the value of the cache that corresponds to the whole program that is analysed, *i.e.* for the program $[t]^p$, we only keep the entry $C(p)$. In §5.3 we present a solver that computes such a cache as an intermediate result.

We define $G(p) \triangleq \gamma_{\mathsf{clos}} \circ \gamma^\sharp_{\mathsf{clos}} \circ \gamma_{\mathsf{NoEnv}} \circ \gamma_{\mathsf{Cache}}(p)$ and $G_{\mathbb{G}} \triangleq \gamma_{\mathbb{G}} \circ \mathrm{map}_{\gamma^\sharp_{\mathsf{clos}}}$. At the top of the abstraction stack, the following semantic inclusions hold:

Lemma 5.4 (Semantic inclusions for ∇CFA).

$$\begin{array}{l}
A_{\mathsf{EC}}(\!|[\mathsf{x}]^p|\!)_{G_{\mathbb{G}}(\Gamma^\sharp)} \mathrel{\dot{\subseteq}^2} G(p)(\Gamma^\sharp(\mathsf{x})) \\
A_{\mathsf{EC}}(\!|[\lambda^\ell \mathsf{x}.\, t]^p|\!)_{G_{\mathbb{G}}(\Gamma^\sharp)} \mathrel{\dot{\subseteq}^2} G(p)\{\lambda^\ell \mathsf{x}.\, t \mapsto \Gamma^\sharp|_{\mathrm{fv}(\lambda^\ell \mathsf{x}.\, t)}\} \\
A_{\mathsf{EC}}(\!|[[t_1]^{p_1}\ [t_2]^{p_2}]^p|\!)_{G_{\mathbb{G}}(\Gamma^\sharp)} \mathrel{\dot{\subseteq}^2} G(p)(c^{\sharp\prime}_3) \\
\textit{where } A_{\mathsf{EC}}(\!|[t_1]^{p_1}|\!)_{G_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} G(p_1)(c^\sharp_1) \\
\textit{and } A_{\mathsf{EC}}(\!|[t_2]^{p_2}|\!)_{G_{\mathbb{G}}(\Gamma)} \mathrel{\dot{\subseteq}^2} G(p_2)(c^\sharp_2) \\
\textit{and } c^{\sharp\prime}_3 = \textit{if } c^\sharp_1 = \top \textit{ then } \top \textit{ else } \bigcup^{\mathsf{clos}^\sharp}_{c^\sharp \in c^\sharp_1} c^\sharp_3(c^\sharp) \\
\textit{where } \forall c^\sharp = \langle \lambda^\ell \mathsf{x}.\, [t_3]^{p_3}, \Gamma^\sharp_3 \rangle \in c^\sharp_1, \\
\qquad\qquad A_{\mathsf{EC}}(\!|[t_3]^{p_3}|\!)_{G_{\mathbb{G}}(\Gamma^\sharp_3, \mathsf{x}: c^\sharp_2)} \mathrel{\dot{\subseteq}^2} G(p_3)(c^\sharp_3(c^\sharp))
\end{array}$$

The inclusions resemble a big-step evaluator that computes sets of values. For variables, a lookup is performed in the environment. For functions, a singleton containing an abstract closure is returned. When functions are found at an application point, each of them is analysed, and the union of their analysis results is returned.

5.3 A Widening-Based Control Flow Analysis

In this section, we give some elements about the implementation of ∇CFA, that is publicly available as a companion artefact. The core of the analyser is concentrated in the following function, named analyze:

Definition 5.5 (∇-analysis).

$$\begin{array}{l}
\mathrm{analyze}(\mathrm{analyze}_{\mathrm{rec}})(\mathrm{iw}, \delta, \Gamma^\sharp, [\mathsf{x}]^p) \triangleq \Gamma^\sharp(\mathsf{x}) \\
\mathrm{analyze}(\mathrm{analyze}_{\mathrm{rec}})(\mathrm{iw}, \delta, \Gamma^\sharp, [\lambda^\ell \mathsf{x}.\, t]^p) \triangleq \{\langle \lambda^\ell \mathsf{x}.t, \Gamma^\sharp|_{\mathrm{fv}(\lambda^\ell \mathsf{x}.\, t)} \rangle\} \\
\mathrm{analyze}(\mathrm{analyze}_{\mathrm{rec}})(\mathrm{iw}, \delta, \Gamma^\sharp, [[t_1]^{p_1}\ [t_2]^{p_2}]^p) \triangleq \\
\quad \mathrm{let}\ V_1 = \mathrm{analyze}_{\mathrm{rec}}(\mathsf{false}, \delta, \Gamma^\sharp, [t_1]^{p_1})\ \mathrm{in} \\
\quad \mathrm{if}\ V_1 = \top\ \mathrm{then}\ \top\ \mathrm{else} \\
\quad \mathrm{let}\ V_2 = \mathrm{analyze}_{\mathrm{rec}}(\mathsf{false}, \delta, \Gamma^\sharp, [t_2]^{p_2})\ \mathrm{in} \\
\quad \mathrm{let}\ \mathrm{iw}' = \mathrm{maximal}(\phi(\delta p))\ \mathrm{in} \\
\quad \bigcup^{\mathsf{clos}^\sharp}_{\langle \lambda^\ell \mathsf{x}.\, [t_3]^{p_3}, \Gamma^\sharp_3 \rangle \in V_1} \mathrm{analyze}_{\mathrm{rec}}(\mathrm{iw}', \phi(\delta p), (\Gamma^\sharp_3, \mathsf{x} : V_2), [t_3]^{p_3}) \\
\mathrm{analyzer} : (\mathrm{bool} \times \mathbb{D} \times \mathbb{G}^\sharp \times \mathcal{T}) \to \mathsf{Clos}^\sharp \triangleq \mu(\mathrm{analyze})
\end{array}$$

The program analyze is executed using a top-down solver, *i.e.* a higher-order function μ of type $((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$, that computes a function $\mu(f)$ such that $f(\mu(f))(x) \leq \mu(f)(x)$. The functional f is written using the *open recursion* style, that was also highlighted in [4]. The actual computation $\mu(f)(x)$ on some argument x is performed on demand. A simple top-down solver memoizes the calls to f together with the so-far computed outputs, and iterates until it finds a post-fixpoint: it computes a *cache* of the calls, that is similar to the cache structure used in CFAs. For this paper, we implemented a simple, non-optimised top-down solver, that we extended so that *input widening* can be demanded by the client function f using the boolean argument iw, as we explain later in the section. More efficient top-down solvers, where computation is *incremental*, have already been described in the literature [1, 5, 38].

Our analyser is parameterised by a projection function ϕ, that is required to have a finite *orbit*:

Definition 5.6 (Finite orbit function)**.** We say that ϕ has a finite orbit, if there is a set S, called the set of *maximal elements*, such that:

- S is finite, and
- for any sequence $(p_n)_n$ of annotations from the program, the sequence $(\delta_n)_n$ defined by $\delta_0 = \varepsilon$ and $\delta_{n+1} = \phi(\delta_n p_n)$ eventually enters S and then remains in S, *i.e.* there exists n_0 such that for every $n \geq n_0$, $\delta_n \in S$.

The finite orbit property is, for instance, satisfied by $\phi = \lfloor\cdot\rfloor_k$, where the maximal elements are the paths of length k. Other choices for ϕ are possible, such as the function that keeps the first elements of a path δ until an element with k occurrences is found, and removes the subsequent elements. We name this strategy k^* in the remainder of the document. As a counter-example, the identity function does not satisfy the finite orbit property.

The function named maximal in our code tests whether an element belongs to the set of maximal elements of ϕ.

The argument iw of the analyser controls whether *input widening* must be performed by the solver. When confronted with a recursive call $\text{analyze}_{\text{rec}}(\text{true}, \delta, \Gamma^\sharp, t)$—*i.e.* when input widening is *demanded* by the client—the top-down solver searches for previous calls with the arguments δ and t and $\Gamma^{\sharp\prime}$, and computes the widening of those $\Gamma^{\sharp\prime}$, together with $\Gamma^\sharp$. This new, widened environment, is then used as argument to actually perform the recursive call.

The termination of the algorithm is ensured by the combination of two facts. First, the top-down solver performs widening for the *outputs* of the function. To keep the results accurate enough, widening on outputs is performed only when the solver detects a cycle; the domain's union is used otherwise. Second, the *inputs* are widened when iw = true. Thus, it is the responsibility of the client function—here, the function analyze—to ensure that enough widenings on inputs are demanded to ensure convergence. Therefore, the termination of the analysis is obtained thanks to the cooperation between the top-down solver and its client function.

The first two recursive calls on the sub-terms $[t_1]^{p_1}$ and $[t_2]^{p_2}$ make the term argument decrease, and are thus not a source of divergence. The only source of non-termination are the recursive calls on the body of the computed closures. There are two cases for such calls: if $\phi(\delta p)$ is not maximal, then the number of annotations p' that can be added before the sequence of call sites reaches the maximal set decreases. This ensures convergence in this case. Otherwise, $\phi(\delta p)$ is maximal, and remains in the set of maximal elements, that is finite. Widening on $\Gamma^\sharp$ is performed, and the stationarity of widening sequences implies the termination of the analyser.

The call site paths that we use in the analyser are only used as elements of a *heuristic* to guide the widening strategy. They are not part of the abstract values, as it is the case in k-CFA. Thus, these paths need no specific semantic justification for the soundness of the analysis to be proved.

6 Experimental Results

We implemented k-CFA and ∇CFA in OCaml [23], so as to compare their precision and cost. For the comparison between the different analyses to be fair, we removed the computation of the cache from k-CFA (so that only the global environment and the abstract value for the whole program are computed), following what we did in §5. We extended the two analyses to support booleans and integers, so that more examples can be analysed. The prototype analysers and the suite of program tests are publicly available in the artefact that accompanies the article.

With these experiments, we want to show that the area of the CFA design space we have uncovered is worth exploring more thoroughly: the experimental results that we obtain on standard examples extracted from the literature [8, 14, 18, 22, 27, 42] are encouraging us in pursuing in this research direction. Our goal is *not* to show that ∇CFA is better than existing CFAs, and more realistic benchmarks, *e.g.* measures on the impact of optimisations for compilers for functional languages, would be required to do so.

We provide in Table 1 an experimental comparison of the standard 0-CFA, the ϕ-CFA with the two strategies of limiting paths to the last call site (1-CFA) and the one limiting to at most one occurrence of call sites (1^*-CFA). We also compare with our widening-based analysis ∇CFA, using the same two strategies (1-∇CFA and 1^*-∇CFA). We are thus able to observe both the impact of the choice of ϕ, and of the choice of standard CFA versus ∇CFA.

We perform the following measurements: we count how many states are needed for the top-down solver to explore the abstract call graph (column "S"), and how many edges this call graph contains (column "E"). These two measures are independent of the quality of our implementation: this graph remains the same, however incremental or efficient the solver is. The third measure is the number of iterations that are needed to find a post-fixpoint (column "Iter"). Because our solver does not try to avoid unnecessary recomputations, this measure can be seen as an upper bound of the iterations that an optimised solver would perform. Lastly, the fourth measure (column "Result") is the obtained abstract value, that we compare for precision, taking 1-CFA as a baseline.

The main takeaways from the experiments are twofold:

1. 1^*-CFA generally yields better results than 1-CFA, because information loss on context sensitivity only happens when *recursion* is involved. For all but one program (church), 1^*-CFA improves over 1-CFA in terms of precision. Moreover, the sizes of the call graphs for 1^*-CFA are generally on par with those of 1-CFA, and so are the number of necessary iterations.
2. 1^*-∇CFA generally yields better results than 1^*-CFA and 1-CFA, in particular on programs that involve arithmetic computations, because classic CFAs cannot exploit expressive numeric domains such as intervals.

Table 1. Experimental results. S: number of states in the call graph. E: edges in the call graph. Iter: number of iterations. R: result of the analysis (B: baseline, =: same, ⊖: less precise, ⊕: more precise, $^\circ$: optimal).

Program	0-CFA				1-CFA				1^*-CFA				1-∇CFA				1^*-∇CFA			
	States	Edges	Iter	Result	States	Edges	Iter	Result	States	Edges	Iter	Result	States	Edges	Iter	Result	States	Edges	Iter	Result
`ack`	35	54	3	=	107	183	4	B	185	321	5	=	177	397	5	⊕	276	579	3	=
`ack_cps`	54	78	5	=	161	250	7	B	187	302	7	=	162	289	3	⊖	292	548	3	⊖
`binomial`	37	59	3	=	99	161	3	B	151	263	4	=	148	274	4	⊕	254	531	4	⊕
`blur`	42	60	6	⊖	64	81	10	B°	93	128	8	$=^\circ$	54	61	2	$=^\circ$	55	61	2	$=^\circ$
`church`	117	251	17	⊖	269	427	21	B	322	446	25	⊖	403	636	3	⊖	444	642	3	⊖
`delta_delta`	6	5	2	$=^\circ$	8	7	3	B°	10	9	4	$=^\circ$	8	7	2	$=^\circ$	10	9	2	$=^\circ$
`eta`	11	12	3	⊖	14	14	3	B°	14	14	3	$=^\circ$	14	14	2	$=^\circ$	14	14	2	$=^\circ$
`facehugger`	37	50	4	=	58	74	5	B	76	102	6	=	76	117	4	⊕	74	99	4	⊕
`fact`	15	19	3	=	25	33	4	B	35	47	4	=	25	39	4	⊕	36	53	4	⊕
`fact_cps`	30	37	7	=	60	71	7	B	65	83	7	=	60	81	2	⊖	76	104	2	⊖
`fact_tailrec`	24	30	5	=	37	47	5	B	49	64	5	=	41	58	3	⊕	50	65	3	⊕
`hmca100`	1307	11902	4	⊖	1605	2002	4	B°	1605	2002	4	$=^\circ$	1605	2002	2	$=^\circ$	1605	2002	2	$=^\circ$
`hmca200`	2607	43802	4	⊖	3205	4002	4	B°	3205	4002	4	$=^\circ$	3205	4002	2	$=^\circ$	3205	4002	2	$=^\circ$
`hmca300`	3907	95702	4	⊖	4805	6002	4	B°	4805	6002	4	$=^\circ$	4805	6002	2	$=^\circ$	4805	6002	2	$=^\circ$
`hmca400`	5207	167602	4	⊖	6405	8002	4	B°	6405	8002	4	$=^\circ$	6405	8002	2	$=^\circ$	6405	8002	2	$=^\circ$
`hmca500`	6507	259502	4	⊖	8005	10002	4	B°	8005	10002	4	$=^\circ$	8005	10002	2	$=^\circ$	8005	10002	2	$=^\circ$
`hmca600`	7807	371402	4	⊖	9605	12002	4	B°	9605	12002	4	$=^\circ$	9605	12002	2	$=^\circ$	9605	12002	2	$=^\circ$
`kcfa2`	28	36	5	=	54	75	5	B	91	97	5	$\oplus^\circ$	59	83	2	=	91	97	2	$\oplus^\circ$
`kcfa3`	36	45	6	=	79	119	6	B	167	173	6	$\oplus^\circ$	88	131	2	=	167	173	2	$\oplus^\circ$
`mc91`	16	22	3	=	37	56	4	B	59	90	5	=	40	68	3	⊕	74	125	2	⊕
`mc91_cps`	31	40	5	=	70	96	8	B	68	94	6	=	82	125	3	⊖	92	130	2	⊖
`mc91_tailrec`	34	47	5	=	74	102	5	B	116	181	5	=	104	180	4	⊕	118	193	3	=
`mj09`	28	31	7	=	42	44	7	B	44	44	7	=	43	47	2	⊕	44	44	2	$\oplus^\circ$
`sat`	55	76	11	=	156	278	14	B	102	113	11	$\oplus^\circ$	192	303	2	=	129	166	3	$\oplus^\circ$
`sat3`	49	68	10	=	105	160	12	B	86	94	10	$\oplus^\circ$	140	208	2	=	96	109	2	$\oplus^\circ$
`sat4`	57	78	11	=	158	280	14	B	104	115	11	$\oplus^\circ$	194	305	2	=	131	168	3	$\oplus^\circ$
`shivers`	8	7	3	$=^\circ$	11	10	4	B°	14	13	5	$=^\circ$	14	18	2	$=^\circ$	17	21	2	$=^\circ$
`shivers2`	31	37	6	$=^\circ$	49	55	8	B°	49	53	9	$=^\circ$	49	56	2	$=^\circ$	49	56	2	$=^\circ$
`tak`	35	59	3	=	137	256	4	B	241	448	5	=	195	401	3	=	402	849	3	=
`tak_cps`	62	92	7	=	227	368	12	B	136	222	9	=	250	388	2	⊖	225	398	2	⊖
`tak_4d`	49	88	3	=	231	464	4	B	421	834	5	=	315	683	3	=	797	1757	3	=

The use of widening generally makes the analysis converge faster but returns a less precise result. The fact that we lose precision with ∇CFA only in a few cases is indication that widening can be used with benefit here.

The `hmca*` examples from [14]—`hmca` stands for Heintze McAllester—show the expected blow-up of 0-CFA. As expected, 1-CFA performs better than 0-CFA on these examples, and gives the most accurate result. Remarkably, the other analyses explore the exact same graphs and therefore give the same results. In particular, the 1^* strategy incurs no performance penalty and ∇CFA does not degrade precision.

One advantage of ∇CFA is that it removes the finiteness constraint on the height of the abstract domain, and thus allows the use of more expressive domains, such as intervals. This benefit is visible in several examples: with ∇CFA, the factorial `fact` is shown to return a result that is greater than 1, and the 91 function of McCarthy `mc91` is shown to produce a result that is above 91. This is the best precision that can be achieved using intervals. This level of precision is not reachable by finite-height domains.

The examples `sat`, `sat3` and `sat4` are boolean satisfiability testers, that are applied to a boolean formula encoded as a function. Both 1-CFA and 1-∇CFA fail to achieve the best result because the depths of the call stacks is statically bounded. In contrast, 1^*-CFA and 1^*-∇CFA compute the best result—*i.e.* the formula is satisfiable—because the call stack is only bounded for *recursively* called program points.

The programs `ack_cps`, `mc91_cps`, `tak_cps` recursively produce a continuation—a higher-order value. Classic CFA is able to show that the three examples produce integers. ∇CFA analyses, however, fail to keep information on the results of the continuations, and are not able to prove that the outputs must be integers. This is due to the fact that our abstract domain for closures cannot represent recursively defined sets of closures, whereas the indirections used in k-CFA can express such solutions. Exploring ways to recover more precision is left for future work. We could for instance perform a narrowing pass, or design a more expressive domain for closures, that keeps information about the results of functions—a sort of function summary—or use a relational domain such as the one of [30], that keeps relations between inputs and outputs of functions.

7 Related Work

Control flow analysis for the lambda calculus was developed by Jones [19]. Shivers [41, 42] proposed a control flow analysis for Scheme programs in continuation-passing style and introduced the notion of *k-CFA* in which flow information is given relative to a context which represents an abstraction of the state. Sestoft [39] developed an analysis for computing the flow of closures in higher-order functional programs. The constraint-based formulation of CFA was introduced by Palsberg [34] and developed further by Nielson and Nielson [31–33]. In their *infinitary control flow analysis*, they propose a constraint-based framework from which they show how to obtain different kinds of k-CFA by varying the notion of context. In that article, they leave it as an open problem to relate the infinitary CFA to a collecting semantics of an operational semantics using abstract interpretation.

Shivers [42] outlines the proof of correctness of a control flow analysis for a CPS-based intermediate representation of Scheme. The proof is based on a non-standard denotational semantics which, in addition to the program's result, computes a *cache* for each call site. This cache contains the identifiers of the lambda expressions called at that call site. From this semantics, it is possible to obtain 0-CFA and 1-CFA as abstract interpretations.

Jones and Rosendahl [21] give a denotational semantics for a higher-order functional language based on *minimal function graphs* [20]. This semantics computes a subset of the graph of a function (*i.e.* a set of (argument,result) pairs) that are necessary and sufficient to evaluate a given function call. This semantics is then abstracted in order to obtain a closure analysis that approximates the control flow of the program. The minimal function graphs could be obtained in our approach, by exploiting the return events that are labelled with call annotations—these events were removed by the first abstraction steps of §3.

Midtgaard and Jensen show how to use the principles of abstract interpretation to systematically derive control flow analyses from collecting semantics induced by abstract machines. They obtained *context-insensitive* analyses—akin to 0-CFA—for continuation-passing style [26] and for direct-style programs in ANF [27] based on the CESK machine.

Van Horn and Might [15] apply abstract interpretation to the internal state representation of a series of abstract state machines. This results in "abstract abstract machines" that compute an over-approximation of control flow and other intensional information about an execution. In machines with store-allocated continuations, control flow analysis becomes the problem of abstracting the store.

Darais *et. al.* [4] demonstrate that the "abstracting abstract machines" approach carry over to more high-level *definitional interpreters* that are not expressed as state machines. In particular, they show that this approach can reconstruct the CFA2 control flow analyses [44, 45] based on push-down structures that also computes precise return information.

Trace partitioning [12, 24, 36] is a technique that can increase the precision of analyses, by exploiting the execution traces of a program to create a partition of its abstract states. We can view the call site paths used as keys in the environments and in the caches of k-CFA as a mean to partition the abstract states: k-CFA computes an environment and a cache for every call site path. We obtained these paths as abstractions of the traces, which is consistent with the trace partitioning approach. Our functions ϕ identify several contexts, and we use them to define context sensitivity policies. A similar approach is followed in [37, Chap. 8.2], where *instances*—timestamps that are used to differentiate instances of formal parameters as in the AAM approach—are abstracted using Galois connections.

8 Conclusions and Future Work

We have proposed a substitution-based small-step operational semantics for the λ-calculus, that produces a trace of the control flow events that take place during reduction. We have devised a series of abstractions which, when applied to the trace-based semantics, derive k-CFA by abstract interpretation. Based on the obtained abstract interpretation, we have produced a semantic justification for the constraint-based presentation of k-CFA.

We have also employed abstract interpretation to define the ∇CFA analysis, a novel control-flow analysis that utilises non-finite domains, and takes advantage of widening to ensure termination of the analyser. We have shown that ∇CFA can compute more precise results than those provided by k-CFA analyses, especially when numeric domains are involved. Our experiments show that using widening in CFAs is not only possible but also brings benefits.

Using the same approach, we think m-CFA [10, 29] could be obtained using a different abstraction on traces, that would keep *frames* instead of call sites. The question remains open whether we could also derive the more recent pushdown CFAs [6, 7, 11, 44, 45]. This could be achieved possibly by employing more precise abstractions. For example, we have not exploited so far the *order* in which events occur, nor have we used the contents of *call* annotations, that record in the calling contexts which functions were called and on which arguments. This extra information is likely to help us justify the well-bracketed aspect of pushdown analyses.

While the experimental results for ∇CFA are encouraging concerning precision, further study is necessary to measure how the analysis performs on larger, real-world programs, and how it compares to pushdown-based analyses. To answer these questions, we need to improve the performance of our analyser. A natural target for improvement is to make our top-down solver incremental, so that unnecessary recomputations are avoided. Another improvement would be to use a more expressive abstract domain for closures, that keeps information on function results and would help the widening operator lose less information.

We have based our semantic development on a semantics of traces. This idea could be pushed further, to analyse other intentional properties of programs, such as effect analysis, or complexity analysis. Moreover, we could also enrich the traces with the inputs of the program—an approach similar to the analysis of input-output relations [30]. This would result in a *relational* control-flow analysis that detects which functions are called at a given program point, depending on the inputs of the program.

References

[1] Baudouin L Charlier and Pascal Van Hentenryck. 1992. *A Universal Top-Down Fixpoint Algorithm*. Technical Report. USA. ftp://ftp.cs.brown.edu/pub/techreports/92/cs92-25.pdf

[2] Patrick Cousot. 1997. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract). *Electronic Notes in Theoretical Computer Science* 6, 1-2 (apr 1997), 77–102. https://doi.org/10.1016/s1571-0661(05)80168-9

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[4] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

[5] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: verifying a local generic solver in Iris. *Proc. ACM Program. Lang.* 4, POPL (2020), 33:1–33:28. https://doi.org/10.1145/3371101

[6] Christopher Earl, Matthew Might, and David Van Horn. 2010. Pushdown Control-Flow Analysis of Higher-Order Programs. *Workshop on Scheme and Functional Programming* abs/1007.4268 (2010). http://arxiv.org/abs/1007.4268

[7] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective pushdown analysis of higher-order programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 177–188. https://doi.org/10.1145/2364527.2364576

[8] Leandro Facchinetti, Zachary Palmer, and Scott F. Smith. 2019. Higher-order Demand-driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 3 (2019), 14:1–14:53. https://doi.org/10.1145/3310340

[9] Kimball Germane and Michael D. Adams. 2020. Liberate Abstract Garbage Collection from the Stack by Decomposing the Heap. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 197–223. https://doi.org/10.1007/978-3-030-44914-8_8

[10] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 407–420. https://doi.org/10.1145/2951913.2951936

[11] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 691–704. https://doi.org/10.1145/2837614.2837631

[12] Maria Handjieva and Stanislav Tzolovski. 1998. Refining Static Analyses by Trace-Based Partitioning Using Control Flow. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1503)*, Giorgio Levi (Ed.). Springer, 200–214. https://doi.org/10.1007/3-540-49727-7_12

[13] Nevin Heintze. 1994. Set-Based Analysis of ML Programs. *ACM SIGPLAN Lisp Pointers* VII, 3 (July 1994), 306–317. https://doi.org/10.1145/182590.182495

[14] Nevin Heintze and David McAllester. 1997. On the Complexity of Set-Based Analysis. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 150–163. https://doi.org/10.1145/258948.258963

[15] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10*. ACM Press. https://doi.org/10.1145/1863543.1863553

[16] Suresh Jagannathan and Stephen Weeks. 1995. A Unified Treatment of Flow Analysis in Higher-order Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press. https://doi.org/10.1145/199448.199536

[17] Suresh Jagannathan and Andrew K. Wright. 1996. Flow-Directed Inlining. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 193–205. https://doi.org/10.1145/231379.231417

[18] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing abstract abstract machines. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP '13*. ACM Press. https://doi.org/10.1145/2500365.2500604

[19] Neil D. Jones. 1981. Flow analysis of lambda expressions. In *Automata, Languages and Programming*, S. Even and O. Kariv (Eds.). Springer Berlin Heidelberg, 114–128. https://doi.org/10.1007/3-540-10843-2_10

[20] Neil D. Jones and Alan Mycroft. 1986. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '86*. ACM Press. https://doi.org/10.1145/512644.512672

[21] Neil D. Jones and Mads Rosendahl. 1997. Higher-Order Minimal Function Graphs. *Journal of Functional and Logic Programming* 1997, 2 (Feb. 1997). http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1997/A97-02/A97-02.html

[22] Donald E Knuth. 1991. Textbook examples of recursion. *Artificial Intelligence and Mathematical Theory of Computation. Papers in Honor of John McCarthy* (1991), 207–229. https://arxiv.org/abs/cs/9301113

[23] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2017. *The Objective Caml System, Documentation and User's Manual – Release 4.06.* INRIA. http://caml.inria.fr/pub/docs/manual-ocaml-4.06/

[24] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 5–20. https://doi.org/10.1007/978-3-540-31987-0_2

[25] Jan Midtgaard. 2012. Control-Flow Analysis of Functional Programs. *Comput. Surveys* 44, 3 (June 2012), 1–33. https://doi.org/10.1145/2187671.2187672

[26] Jan Midtgaard and Thomas P. Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Static Analysis, 15th International Symposium, SAS 2008*, María Alpuente and Germán Vidal (Eds.). Springer LNCS vol. 5079, 347–362. https://doi.org/10.1007/978-3-540-69166-2_23

[27] Jan Midtgaard and Thomas P. Jensen. 2009. Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09*. ACM Press. https://doi.org/10.1145/1596550.1596592

[28] Matthew Might and Olin Shivers. 2006. Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/1159803.1159807

[29] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 305–315. https://doi.org/10.1145/1806596.1806631

[30] Benoît Montagu and Thomas P. Jensen. 2020. Stable Relations and Abstract Interpretation of Higher-order Programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 119:1–119:30. https://doi.org/10.1145/3409001

[31] Flemming Nielson and Hanne Riis Nielson. 1997. Infinitary Control Flow Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*. ACM Press. https://doi.org/10.1145/263699.263745

[32] Flemming Nielson and Hanne Riis Nielson. 2006. Types from Control Flow Analysis. In *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 4444)*, Thomas W. Reps, Mooly Sagiv, and Jörg Bauer (Eds.). Springer, 293–310. https://doi.org/10.1007/978-3-540-71322-7_14

[33] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-03811-6

[34] Jens Palsberg. 1995. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems* 17, 1 (jan 1995), 47–62. https://doi.org/10.1145/200994.201001

[35] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.

[36] Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26. https://doi.org/10.1145/1275497.1275501

[37] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective*. The MIT Press, Cambridge, Massachusetts.

[38] Helmut Seidl and Ralf Vogler. 2018. Three Improvements to the Top-Down Solver. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, David Sabel and Peter Thiemann (Eds.). ACM, 21:1–21:14. https://doi.org/10.1145/3236950.3236967

[39] Peter Sestoft. 1989. Replacing function parameters by global variables. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*. ACM Press, 39–53. https://doi.org/10.1145/99370.99374

[40] Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (Eds.). Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 7, 189–233.

[41] Olin Shivers. 1988. Control-Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation - PLDI '88*, Richard L. Wexelblat (Ed.). ACM, 164–174. https://doi.org/10.1145/53990.54007

[42] Olin Shivers. 1991. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*. Association for Computing Machinery, New York, NY, USA, 190–198. https://doi.org/10.1145/115865.115884

[43] David Van Horn and Harry G. Mairson. 2008. Deciding kCFA is Complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 275–282. https://doi.org/10.1145/1411204.1411243

[44] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *Proc. of 19th European Symposium on Programming, ESOP 2010*. Springer LNCS vol. 6012, 570–589. https://doi.org/10.1007/978-3-642-11957-6_30

[45] Dimitrios Vardoulakis and Olin Shivers. 2011. Pushdown flow analysis of first-class control. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 69–80. https://doi.org/10.1145/2034773.2034785

[46] Andrew K. Wright and Suresh Jagannathan. 1998. Polymorphic Splitting: An Effective Polyvariant Flow Analysis. *ACM Trans. Program. Lang. Syst.* 20, 1 (1998), 166–207. https://doi.org/10.1145/271510.271523